Table of Contents

Principles and Techniques of Data Science

By Sam Lau, Joey Gonzalez, and Deb Nolan.

This is the textbook for Data 100, the Principles and Techniques of Data Science course at UC Berkeley.

DS100 is the upper-division data science course intended to be taken after Data 8, the Foundations of Data Science course.

If you are interested in setting up the textbook for local development, see the the setup guide.

The Data Science Lifecycle

In data science, we use large and diverse data sets to make conclusions about the world. In this course, we will discuss the key principles and techniques of data science through the dual lens of computational and inferential thinking. Practically speaking, this involves the following process:

1. Formulating a question or problem
2. Acquiring and cleaning data
3. Conducting exploratory data analysis
4. Using prediction and inference to draw conclusions

It is quite common for more questions and problems to emerge after the last step of this process, and we can thus repeatedly engage in this procedure to discover new characteristics of our world. This positive feedback loop is so central to our work that we call it the **data science lifecycle**.

If the data science lifecycle were as easy to conduct as it is to state, there would be no need for textbooks about the the subject. Fortunately, each of the steps in the lifecycle contain numerous challenges that reveal powerful and often surprising insights that form the foundation of making thoughtful decisions using data.

As in Data 8, we will begin with an example.

About This Book

Before we continue, it is important for us to state our assumptions about the reader.

In this book, we will proceed as though you have taken Data 8 or some equivalent. In particular, we will assume that you have some familiarity with the following topics (links to pages from the Data 8 textbook are given in parentheses).

- Tabular data manipulation: selection, filtering, grouping, joining (link)
- Sampling, empirical distributions of statistics (link)
- Hypothesis testing using bootstrap resampling (link)
- Least squares regression and regression inference (link)
- Classification (link)

In addition, we will assume that you have taken CS61A or some equivalent and thus will not explain Python syntax except in special cases.

Show Widgets Open on DataHub

Table of Contents

The Students of Data 100¶

Recall that the data science lifecycle involves the following broad steps:

1. **Question/Problem Formulation:**
   i. What do we want to know or what problems are we trying to solve?
   ii. What are our hypotheses?
   iii. What are our metrics of success?

2. **Data Acquisition and Cleaning:**
   i. What data do we have and what data do we need?
   ii. How will we collect more data?
   iii. How do we organize the data for analysis?

3. **Exploratory Data Analysis:**
   i. Do we already have relevant data?
   ii. What are the biases, anomalies, or other issues with the data?
   iii. How do we transform the data to enable effective analysis?

4. **Prediction and Inference:**
   i. What does the data say about the world?
   ii. Does it answer our questions or accurately solve the problem?
   iii. How robust are our conclusions?

Question Formulation¶

We would like to figure out if the data we have on student names in Data 100 give us any additional information about the students themselves. Although this is a vague question to ask, it is enough to get us working with our data and we can surely make the question more

precise as we go.

Data Acquisition and Cleaning¶

In Data 100, we will study various methods to collect data.

Let's begin by looking at our data, the roster of student first names that we've downloaded from a previous offering of Data 100.

Don't worry if you don't understand the code for now; we'll introduce the libraries in more depth later. Instead, focus on the process and the charts that we show.

```
import pandas as pd

students = pd.read_csv('roster.csv')
students
```

	Name	Role
0	Keeley	Student
1	John	Student
2	BRYAN	Student
...	...	...
276	Ernesto	Waitlist Student
277	Athan	Waitlist Student
278	Michael	Waitlist Student

279 rows × 2 columns

We can quickly see that there are some quirks in the data. For example, one of the student's names is all uppercase letters. In addition, it is not obvious what the Role column is for.

In Data 100, we will study how to identify anomalies in data and apply corrections. The differences in capitalization will cause our programs to think that `'BRYAN'` and `'Bryan'` are different names when they are identical for our purposes. Let's convert all names to lower case to avoid this.

```
students['Name'] = students['Name'].str.lower()
students
```

	Name	Role
0	keeley	Student
1	john	Student
2	bryan	Student
...	...	...
276	ernesto	Waitlist Student
277	athan	Waitlist Student
278	michael	Waitlist Student

279 rows × 2 columns

Now that our data are in a format that's easier for us to work with, let's proceed to exploratory data analysis.

Show Widgets Open on DataHub

Table of Contents

Exploratory Data Analysis¶

The term Exploratory Data Analysis (EDA for short) refers to the process of discovering traits about our data that inform future analysis.

Here's the `students` table from the previous page:

```
students
```

	Name	Role
0	keeley	Student
1	john	Student
2	bryan	Student
...	...	...
276	ernesto	Waitlist Student
277	athan	Waitlist Student
278	michael	Waitlist Student

279 rows × 2 columns

We are left with a number of questions. How many students are in this roster? What does the `Role` column mean? We conduct EDA in order to understand our data more thoroughly.

In Data 100 we will study exploratory data analysis and practice analyzing new datasets.

Oftentimes, we explore the data by repeatly posing simple questions about the data that we'd like to know about. We will structure our analysis this way.

How many students are in our dataset?

```
print("There are", len(students), "students on the roster.")
```

```
There are 279 students on the roster.
```

A natural follow-up question is whether this is the complete list of students or not. In this case, we happen to know that this list contains all students in the class.

What is the meaning of the `Role` field?

Understanding the meaning of field can often be achieved by looking at the unique values of the field's data:

```
students['Role'].value_counts().to_frame()
```

	Role
Student	237
Waitlist Student	42

We can see here that our data contain not only students enrolled in the class at the time but also the students on the waitlist. The `Role` column tells us whether each student is enrolled.

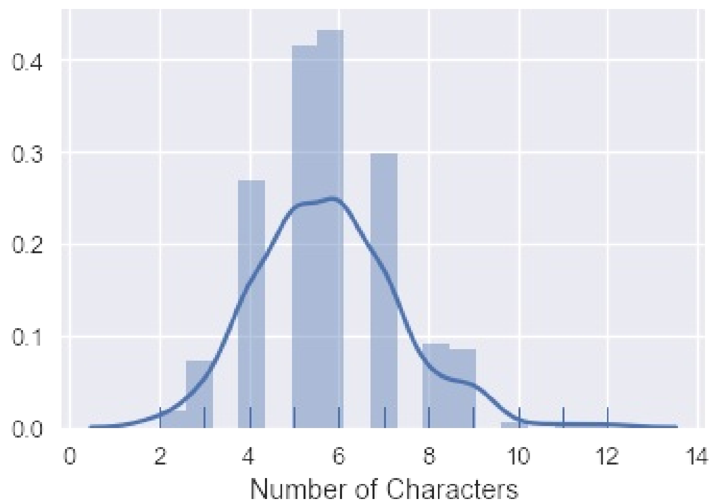
What about the names? How can we summarize this field?

In Data 100 we will deal with many different kinds of data (not just numbers) and we will study techniques to diverse types of data.

A good starting point might be to examine the lengths of the strings.

```
sns.distplot(students['Name'].str.len(), rug=True,
axlabel="Number of Characters")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10e6fd0b8>
```



This visualization shows us that most names are between 3 and 9 characters long. This gives us a chance to check whether our data seem reasonable — if there were many names that were 1 character long we'd have good reason to re-examine our data.

What's in a Name?¶

Although this dataset is rather simple, we will soon see that first names alone can reveal quite a bit about our class.

Show Widgets Open on DataHub

Table of Contents

What's in a Name?¶

So far, we have asked a broad question about our data: "Do the first names of students in Data 100 tell us anything about the class?"

We have done a bit of data cleaning by converting all our names to lowercase. During our exploratory data analysis we discovered that our roster contains about 270 names of students in the class and on the waitlist, and that most of our first names are between 4 and 8 characters long.

What else can we discover about our class based on first names? We might consider a single name from our dataset:

```
students['Name'][5]
```

```
'jerry'
```

From this name we can infer that the student is likely a male. We can also take a guess at the student's age. For example, if we happen to know that Jerry was a very popular baby name in 1998, we might guess that this student is around twenty years old.

This thinking gives us two new questions to investigate:

1. "Do the first names of students in Data 100 tell us the distribution of sex in the class?"
2. "Do the first names of students in Data 100 tell us the distribution of ages in the class?"

In order to investigate these questions, we will need a dataset that associates names with sex and year. Conveniently, the US Social Security department hosts such a dataset online: https://www.ssa.gov/oact/babynames/index.html. Their dataset records the names given to babies at birth and as such is often referred to as the Baby Names dataset.

We will start by downloading and then loading the dataset into Python. Again, don't worry about understanding the code in this first chapter. It's more important that you understand the overall process.

```
import urllib.request
import os.path

data_url = "https://www.ssa.gov/oact/babynames/names.zip"
local_filename = "babynames.zip"
if not os.path.exists(local_filename): # if the data exists
don't download again
    with urllib.request.urlopen(data_url) as resp,
open(local_filename, 'wb') as f:
        f.write(resp.read())

import zipfile
babynames = []
with zipfile.ZipFile(local_filename, "r") as zf:
    data_files = [f for f in zf.filelist if f.filename[-3:] ==
"txt"]
    def extract_year_from_filename(fn):
        return int(fn[3:7])
    for f in data_files:
        year = extract_year_from_filename(f.filename)
        with zf.open(f) as fp:
            df = pd.read_csv(fp, names=["Name", "Sex", "Count"])
            df["Year"] = year
            babynames.append(df)
babynames = pd.concat(babynames)
babynames
```

	Name	Sex	Count	Year
0	Mary	F	9217	1884
1	Anna	F	3860	1884
2	Emma	F	2587	1884
...	...	...	...	...
2081	Verna	M	5	1883
2082	Winnie	M	5	1883
2083	Winthrop	M	5	1883

1891894 rows × 4 columns

```
ls -alh babynames.csv
```

```
-rw-r--r--  1 sam  staff    30M Jan 22 15:31 babynames.csv
```

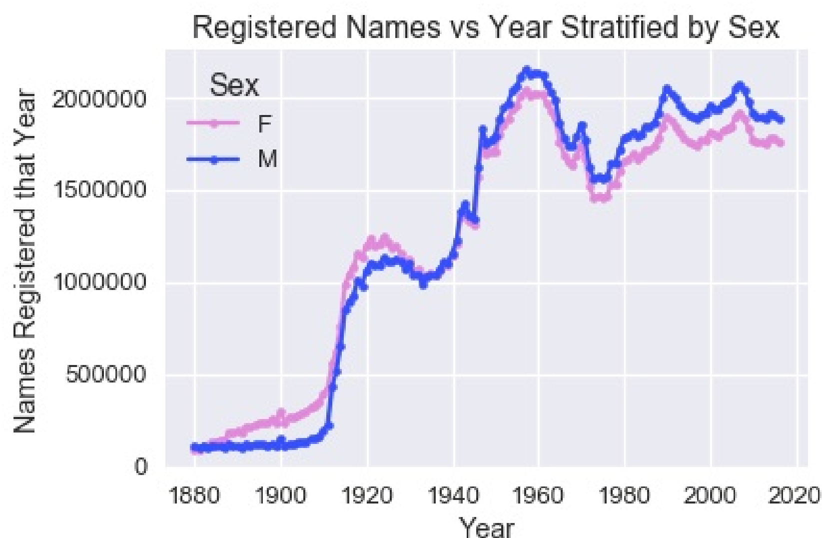
It looks like the dataset contains names, the sex given to the baby, the number of babies with that name, and the year of birth for those babies. To be sure, we check the dataset description from the SSN Office: https://www.ssa.gov/oact/babynames/background.html

> All names are from Social Security card applications for births that occurred in the United States after 1879. Note that many people born before 1937 never applied for a Social Security card, so their names are not included in our data. For others who did apply, our records may not show the place of birth, and again their names are not included in our data.
>
> All data are from a 100% sample of our records on Social Security card applications as of March 2017.

A useful visualization for this data is to plot the number of male and female babies born each year:

```
pivot_year_name_count = pd.pivot_table(
    babynames, index='Year', columns='Sex',
    values='Count', aggfunc=np.sum)

pink_blue = ["#E188DB", "#334FFF"]
with sns.color_palette(sns.color_palette(pink_blue)):
    pivot_year_name_count.plot(marker=".")
    plt.title("Registered Names vs Year Stratified by Sex")
    plt.ylabel('Names Registered that Year')
```



This plot makes us question whether the US had babies in 1880. A sentence from the quote above helps explain:

> Note that many people born before 1937 never applied for a Social Security card, so their names are not included in our data. For others who did apply, our records may not show the place of birth, and again their names are not included in our data.

We can also see the baby boomer period quite clearly in the plot above.

Inferring Sex From Name¶

Let's use this dataset to estimate the number of females and males in our class. As with our class roster, we begin by lowercasing the names:

```
babynames['Name'] = babynames['Name'].str.lower()
babynames
```

	Name	Sex	Count	Year
0	mary	F	9217	1884
1	anna	F	3860	1884
2	emma	F	2587	1884
...	...	...	...	...
2081	verna	M	5	1883
2082	winnie	M	5	1883
2083	winthrop	M	5	1883

1891894 rows × 4 columns

Then, we count up how many male and female babies were born in total for each name:

```
sex_counts = pd.pivot_table(babynames, index='Name',
columns='Sex', values='Count',
                              aggfunc='sum', fill_value=0.,
margins=True)
sex_counts
```

Sex	F	M	All
Name			
aaban	0	96	96
aabha	35	0	35
aabid	0	10	10
...	...	...	...
zyyon	0	6	6
zzyzx	0	5	5
All	170639571	173894326	344533897

96175 rows × 3 columns

To decide whether a name is male or female, we can compute the proportion of times the name was given to a female baby.

```
prop_female = sex_counts['F'] / sex_counts['All']
sex_counts['prop_female'] = prop_female
sex_counts
```

Sex	F	M	All	prop_female
Name				
aaban	0	96	96	0.000000
aabha	35	0	35	1.000000
aabid	0	10	10	0.000000
...	...	...	...	...
zyyon	0	6	6	0.000000
zzyzx	0	5	5	0.000000
All	170639571	173894326	344533897	0.495277

96175 rows × 4 columns

We can then define a function that looks up the proportion of female names given a name.

```
def sex_from_name(name):
    if name in sex_counts.index:
        prop = sex_counts.loc[name, 'prop_female']
        return 'F' if prop > 0.5 else 'M'
    else:
        return None
sex_from_name('sam')
```

```
'M'
```

Try typing some names in this box to see whether the function outputs what you expect:

```
interact(sex_from_name, name='sam');
```

Show Widget
We mark each name in our class roster with its most likely sex.

```
students['sex'] = students['Name'].apply(sex_from_name)
students
```

	Name	Role	sex
0	keeley	Student	F
1	john	Student	M
2	bryan	Student	M
...	...	...	...
276	ernesto	Waitlist Student	M
277	athan	Waitlist Student	M
278	michael	Waitlist Student	M

279 rows × 3 columns

Now it is easy to estimate how many male and female students we have:

```
students['sex'].value_counts()
```

```
M    144
F     92
Name: sex, dtype: int64
```

Inferring Age From Name¶

We can proceed in a similar way to estimate the age distribution of the class, mapping each name to its average age in the dataset.

```
def avg_year(group):
    return np.average(group['Year'], weights=group['Count'])

avg_years = (
    babynames
    .groupby('Name')
    .apply(avg_year)
    .rename('avg_year')
    .to_frame()
)
avg_years
```

	avg_year
Name	
aaban	2012.572917
aabha	2013.714286
aabid	2009.500000
...	...
zyyanna	2010.000000
zyyon	2014.000000
zzyzx	2010.000000

96174 rows × 1 columns

```
def year_from_name(name):
    return (avg_years.loc[name, 'avg_year']
            if name in avg_years.index
            else None)

# Generate input box for you to try some names out:
interact(year_from_name, name='fernando');
```

Show Widget

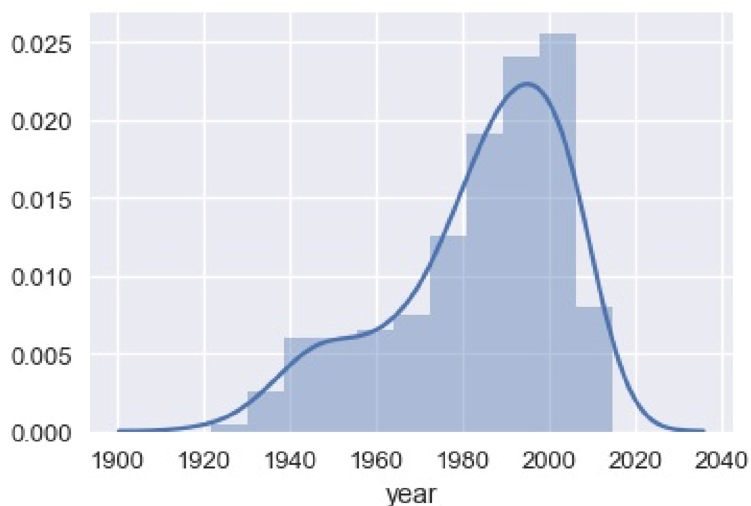
```
students['year'] = students['Name'].apply(year_from_name)
students
```

	Name	Role	sex	year
0	keeley	Student	F	1998.147952
1	john	Student	M	1951.084937
2	bryan	Student	M	1983.565113
...	...	...	...	...
276	ernesto	Waitlist Student	M	1981.439873
277	athan	Waitlist Student	M	2004.397863
278	michael	Waitlist Student	M	1971.179231

279 rows × 4 columns

Then, it is easy to plot the distribution of years:

```
sns.distplot(students['year'].dropna());
```



To compute the average year:

```
students['year'].mean()
```

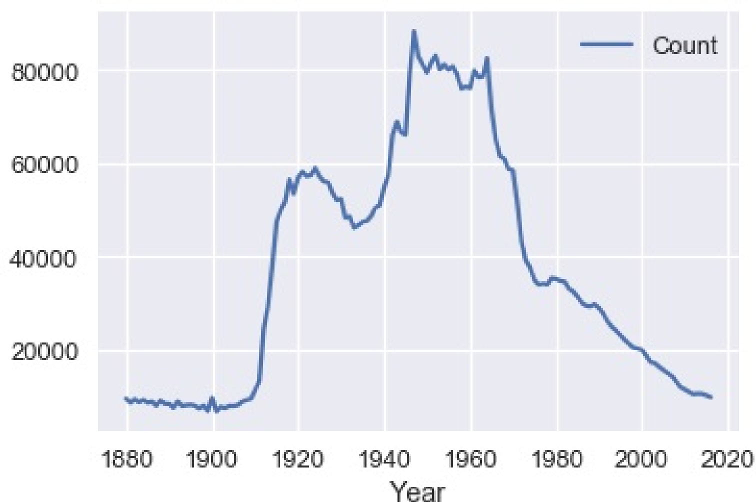
```
1983.846741800525
```

Which makes it appear like the average student is 35 years old. This is a course for college undergrads, so we were expecting an average age of around 20. Why might our estimate be so far off?

As data scientists, we often run into results that don't agree with our expectations and must make judgement calls about whether our results were caused by our data, our processes, or incorrect hypotheses. It is impossible to define rules that apply to every situation. Instead, we will give you the tools to re-examine each step of your data analysis and show you how to use them.

In this case, the most likely possibility for our unexpected result is that most names are old. For example, the name John was quite popular throughout the history recorded in our data, which means that we'd likely guess that a person named John is born around 1950. We can confirm this by looking at the data:

```
names = babynames.set_index('Name').sort_values('Year')
john = names.loc['john']
john[john['Sex'] == 'M'].plot('Year', 'Count');
```



If we happen to believe that no one in our class is over the age of 40 or under the age of 10 (we might find out by looking at our classroom during lecture) we can incorporate this into our analysis by only examining the data between 1978 and 2008. We will soon discuss data manipulation and you may revisit this analysis to find out if incorporating this prior belief gives a more sensible result.

Data Generation

Data science would hardly be a science without data. It is thus of utmost importance that we begin any data analysis by understanding how our data were generated.

In this chapter, we will discuss **data provenance**. Although the term data provenance typically refers to the entire history of data and where it has moved over time, we will use the term in this textbook to refer to the process that generated our data. Many have drawn premature conclusions because they were not careful enough in understanding their data; we will discuss an example of this to justify the importance of probability sampling in conducting data science.

Dewey Defeats Truman

In the 1948 US Presidential election, New York Governor Thomas Dewey ran against the incumbent Harry Truman. As usual, a number of polling agencies conducted polls of voters in order to predict which candidate was more likely to win the election.

1936: A Previous Polling Catastrophe

In 1936, three elections prior to 1948, the *Literary Digest* infamously predicted a landslide defeat for Franklin Delano Roosevelt. To make this claim, the magazine polled a sample of over 2 million people based on telephone and car registrations. As you may know, this sampling scheme suffers from sampling bias: those with telephones and cars tend to be wealthier than those without. In this case, the sampling bias was so great that the *Literary Digest* thought Roosevelt would only receive 43% of the popular vote when he ended up with 61% of the popular vote, a difference of almost 20% and the largest error ever made by a major poll. The *Literary Digest* went out of business soon after [1].

1948: The Gallup Poll

Determined to learn from past mistakes, the Gallup Poll used a method called *quota sampling* to predict the results of the 1948 election. In their sampling scheme, each interviewer polled a set number of people from each demographic class. For example, the interviews were required to interview both males and females from different ages, ethnicities, and income levels to match the demographics in the US Census. This ensured that the poll would not leave out important subgroups of the voting population.

Using this method, the Gallup Poll predicted that Thomas Dewey would earn 5% more of the popular vote than Harry Truman would. This difference was significant enough that the *Chicago Tribune* famously printed the headline "Dewey Defeats Truman":



As we know now, Truman ended up winning the election. In fact, he won with 5% more of the popular vote than Dewey! What went wrong with the Gallup Poll?

The Problem With Quota Sampling

Although quota sampling did help pollsters reduce sampling bias, it introduced bias in another way. The Gallup Poll told its interviewers that as long as they fulfilled their quotas they could interview whomever they wished. Here's one possible explanation for why the interviewers ended up polling a disproportionate number of Republicans: at the time, Republicans were on average wealthier and more likely to live in nicer neighborhoods, making them easier to interview. This observation is supported by the fact that the Gallup Poll predicted 2-6% more Republican votes than the actual results for the 3 elections prior.

These examples highlight the importance of understanding sampling bias as much as possible during the data collection process. Both *Literary Digest* and Gallup Poll made the mistake of assuming their methods were unbiased when their sampling schemes were based on human judgement all along.

We now rely on **probability sampling**, a family of sampling methods that assigns precise probabilities to the appearance of each sample, to reduce bias as much as possible in our data collection process.

Big Data?

In the age of Big Data, we are tempted to deal with bias by collecting more data. After all, we know that a census will give us perfect estimates; shouldn't a very large sample give almost perfect estimates regardless of the sampling technique?

We will return to this question after discussing probability sampling methods to compare the two approaches.

1

> 1. https://www.qualtrics.com/blog/the-1936-election-a-polling-catastrophe/ ↩

Probability Sampling

Unlike convenience sampling, probability sampling allows us to assign a precise probability to the event that we draw a particular sample. We will begin by reviewing simple random samples from Data 8, then introduce two alternative methods of probability sampling: cluster sampling and stratified sampling.

Suppose we have a population of 6 individuals. We've given each individual a different letter from $A-F$.

Simple Random Sample (SRS)

To take an simple random sample of size 2 from this population, we can write each letter $A-F$ on a single index card, place all the cards into a hat, mix the cards well, and draw 2 cards without looking. That is, a SRS is sampling uniformly at random without replacement.

Here are all possible samples of size 2:

$$ \begin{matrix} AB & BC & CD & DE & EF\\ AC&BD&CE&DF&\\ AD&BE&CF&&\\ AE&BF&&&\\ AF&&&&\\ \end{matrix} $$

There are 15 possible samples of size 2 from our population of 6. Another way to count the number of possible samples is:

$$ \binom{6}{2} = \frac{6!}{2!4!} = 15 $$

Since in a SRS we sample uniformly at random, each of these 15 samples are equally likely to be chosen:

$$ P(AB) = P(CD) = \ldots = P(DF) = \frac{1}{15} $$

We can also use this chance mechanism to answer other questions about the composition of the sample. For example:

$$ P(A \text{ in sample}) = \frac{5}{15} = \frac{1}{3}$$

Since 5 out of 15 of the possible samples listed above contain A.

By symmetry, we can say:

$$ P(A \text{ in sample}) = P(F \text{ in sample}) = \frac{1}{3} $$

Another way of computing $ P(A \text{ in sample}) $ is to recognize that for A to be in the sample, we either need to draw it as the first marble or as the second marble.

$$\begin{aligned} P(A \text{ in sample}) &= P(\text{A is first or A is second}) \\ &= P(\text{A is first}) + P(\text{A is second}) \\ &= \frac{1}{6} \cdot \frac{5}{5} + \frac{5}{6} \cdot \frac{1}{5} \\ &= \frac{1}{3} \\ \end{aligned}$$

Cluster Sampling

In cluster sampling, we divide the population into clusters. Then, we use SRS to select clusters at random instead of individuals.

As an example, suppose we take our population of 6 individuals and we pair each of them up: $ (A,B)\quad (C,D) \quad (E,F) $ to form 3 clusters of 2 individuals. Then, we use SRS to select one cluster to produce a sample of size 2.

As before, we can compute the probability that A is in our sample:

$$P(A \text{ in sample}) = P(AB \text{ drawn}) = \frac{1}{3}$$

Similarly the probability that any particular person appears in our sample is $\frac{1}{3}$. Note that this is the same as our SRS. However, we see differences when we look at the samples themselves. For example, in a SRS the chance of getting AB is the same as the chance of getting AC: $\frac{1}{15}$. However, with this cluster sampling scheme:

$$\begin{aligned} P(AB) &= \frac{1}{3} \\ P(AC) &= 0 \end{aligned}$$

Since A and C can never appear in the same sample if we only select one cluster.

Cluster sampling is still probability sampling since we can assign a probability to each potential sample. However, the resulting probabilities are different than using a SRS depending on how the population is clustered.

Why use cluster sampling? Cluster sampling is most useful because it makes sample collection easier. For example, it is much easier to poll towns of 100 people each than to poll thousands of people distributed across the entire US. This is the reason why many polling agencies today use forms of cluster sampling to conduct surveys.

The main downside of cluster sampling is that it tends to produce greater variation in estimation. This typically means that we take larger samples when using cluster sampling. Note that the reality is much more complicated than this, but we will leave the details to a future course on sampling techniques.

Stratified Sampling

In stratified sampling, we divide the population into strata, and then produce one simple random sample per strata. In both cluster sampling and stratified sampling we split the population into groups; in cluster sampling we use a single SRS to select groups whereas in

stratified sampling we use multiple SRS's, one for each group.

We can divide our population of 6 individuals into the following strata:

$$ \begin{matrix} \text{Strata 1:} & {A,B,C,D} \\ \text{Strata 2:} & {E,F} \end{matrix} $$

We use an SRS to select one individual from each strata to produce a sample of size 2. This gives us the following possible samples:

$$ \begin{aligned} (A,E)\quad (A,F) \quad (B,E) \quad (B,F) \quad (C,E) \quad (C,F) \quad (D,E) \quad (D,F) \end{aligned} $$

Again, we can compute the probability that A is in our sample:

$$ \begin{aligned} P(A \text{ in sample}) &= P(A \text{ selected from Strata 1}) \\ &= \frac{1}{4} \end{aligned} $$

However:

$$ \begin{aligned} P(AB) &= 0 \end{aligned} $$

since A and B cannot appear in the same sample.

Like cluster sampling, stratified sampling is also a probability sampling method that produces different probabilities depending on the stratification of the population. Note that like this example, the strata do not have to be the same size. For example, we can stratify the US by occupation, then take samples from each strata of size proportional to the distribution of occupations in the US — if only 0.01% of people in the US are statisticians, we can ensure that 0.01% of our sample will be composed of statisticians. A simple random sample might miss the poor statisticians altogether!

As you may have figured out, stratified sampling can perhaps be called the proper way to conduct quota sampling. It allows the researcher to ensure that subgroups of the population are well-represented in the sample without using human judgement to select the individuals in the sample. This can often result in less variation in estimation. However, stratified sampling is sometimes more difficult to accomplish because we sometimes don't know how large each strata is. In the previous example we have the advantage of the US census, but other times we are not so fortunate.

Why Probability Sampling?

As we have seen in Data 8, probability sampling enables us to quantify our uncertainty about an estimation or prediction. It is only through this precision that we can conduct inference and hypothesis testing. Be wary when anyone gives you p-values or confidence levels without a proper explanation of their sampling techniques.

Now that we understand probability sampling let us see how the humble SRS compares against "big data".

Show Widgets Open on DataHub

Table of Contents

SRS vs. "Big Data"¶

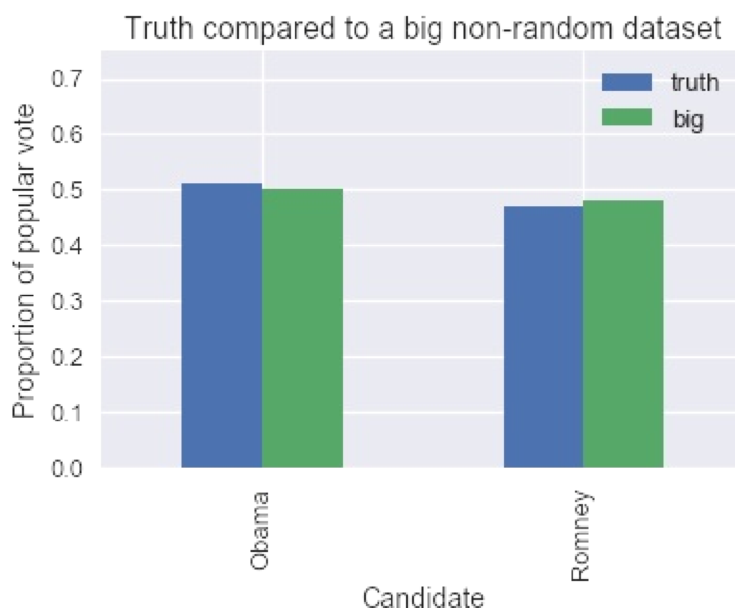
As we have previously mentioned, it is tempting to do away with our long-winded bias concerns by using huge amounts of data. It is true that collecting a census will by definition produce unbiased estimations. Perhaps we don't have to worry about bias if we just collect tons of data.

Suppose we are pollsters in 2012 trying to predict the popular vote of the US presidential election, where Barack Obama ran against Mitt Romney. Since we know the exact output of the popular vote, we can compare the predictions of a SRS to the predictions of a large non-random dataset, often called *administrative datasets* since they are often collected as part of some administrative work.

We will compare a SRS of size 400 to a non-random sample of size 60,000,000. Our non-random sample is nearly 150,000 times larger than our SRS! Since there were about 120,000,000 voters in 2012, we can think of our non-random sample as a survey where half of all voters in the US responded (no actual poll has ever surveyed more than 10,000,000 voters).

Here's a plot comparing the proportions of the non-random sample to the true proportions. The bars labeled `truth` show the true proportions of votes that each candidate received. The bars labeled `big` show the proportions from our dataset of 60,000,000 voters.

```
pd.DataFrame({
    'truth': [obama_true, romney_true],
    'big': [obama_big, romney_big],
}, index=['Obama', 'Romney'], columns=['truth',
'big']).plot.bar()
plt.title('Truth compared to a big non-random dataset')
plt.xlabel('Candidate')
plt.ylabel('Proportion of popular vote')
plt.ylim(0, 0.75)
None
```



We can see that our large dataset is just a bit biased towards the Republican candidate Romney just as the Gallup Poll was in 1948. Still, this dataset could give us accurate predictions. To check, we can simulate taking simple random samples of size 400 from the population and large non-random samples of size 60,000,000. We will compute the proportion of votes for Obama in each sample and plot the distribution of proportions.

```
srs_size = 400
big_size = 60000000
replications = 10000

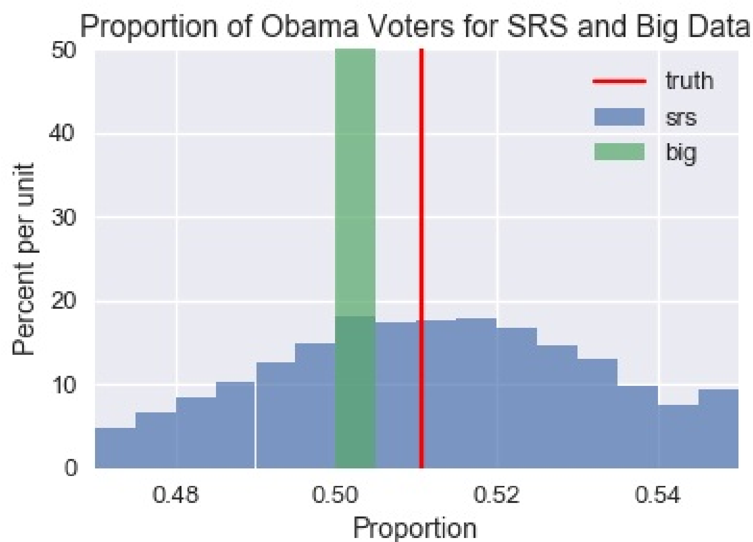
def resample(size, prop, replications):
    return np.random.binomial(n=size, p=prop, size=replications)
/ size

srs_simulations = resample(srs_size, obama_true, replications)
big_simulations = resample(big_size, obama_big, replications)
```

Now, we will plot the simulation results and overlay a red line indicating the true proportion of voters that voted for Obama.

```
bins = bins=np.arange(0.47, 0.55, 0.005)
plt.hist(srs_simulations, bins=bins, alpha=0.7, normed=True,
label='srs')
plt.hist(big_simulations, bins=bins, alpha=0.7, normed=True,
label='big')

plt.title('Proportion of Obama Voters for SRS and Big Data')
plt.xlabel('Proportion')
plt.ylabel('Percent per unit')
plt.xlim(0.47, 0.55)
plt.ylim(0, 50)
plt.axvline(x=obama_true, color='r', label='truth')
plt.legend()
None
```



As you can see, the SRS distribution is spread out but centered around the true population proportion of Obama voters. The distribution created by the large non-random sample, on the other hand, is very narrow but not a single simulated sample produces the true population proportion. If we attempt to create confidence intervals using the non-random sample, none of them will contain the true population proportion. To make matters worse, the confidence interval will be extremely narrow because the sample is so large. We will be very sure of an ultimately incorrect estimation.

In fact, when our sampling method is biased our estimations will become **worse** as we collect more data since we will be more certain about an incorrect result, only becoming more accurate when our dataset is almost a census of our population. **The quality of the data matters much more than its size.**

Takeaways¶

Before accepting the results of a data analysis, it pays to carefully inspect the quality of the data. In particular, we must ask the following questions:

1. Is the data a census (does it include the entire population of interest)? If so, we can just compute properties of the population directly without having to use inference.
2. If the data is a sample, how was the sample collected? To properly conduct inference, the sample should have been collected according to a probability sampling method.
3. What changes were made to the data before producing results? Do any of these changes affect the quality of the data?

For more details on the comparison between random and large non-random samples, we suggest watching this lecture by the statistician Xiao-Li Meng.

Working with Tabular Data

Tabular data, like the datasets we have worked with in Data 8, are one of the most common and useful forms of data for analysis. We introduce tabular data manipulation using `pandas`, the standard Python library for working with tabular data. Although `pandas`'s syntax is more challenging to use than the `datascience` package used in Data 8, `pandas` provides significant performance improvements and is the current tool of choice in both industry and academia for working with tabular data.

It is more important that you understand the types of useful operations on data than the exact details of `pandas` syntax. For example, knowing when to use a group or a join is more useful than knowing how to call the `pandas` function to group data. It is relatively easy to look up the function you need once you know the right operation to use. All of the table manipulations in this chapter will also appear again in a new syntax when we cover SQL, so it will help you to understand them now.

Because we will cover only the most important `pandas` functions in this textbook, you should bookmark the `pandas` documentation for reference when you conduct your own data analyses.

Show Widgets Open on DataHub

Table of Contents

Getting Started¶

In each section of this chapter we will work with the Baby Names dataset from Chapter 1. We will pose a question, break the question down into high-level steps, then translate each step into Python code using `pandas` DataFrames. We begin by importing `pandas` :

```
# pd is a common shorthand for pandas
import pandas as pd
```

Now we can read in the data using `pd.read_csv` (docs).

```
baby = pd.read_csv('babynames.csv')
baby
```

	Name	Sex	Count	Year
0	Mary	F	9217	1884
1	Anna	F	3860	1884
2	Emma	F	2587	1884
...	...	...	...	...
1891891	Verna	M	5	1883
1891892	Winnie	M	5	1883
1891893	Winthrop	M	5	1883

1891894 rows × 4 columns

Note that for the code above to work, the `babynames.csv` file must be located in the same directory as this notebook. We can check what files are in the current folder by running `ls` in a notebook cell:

```
ls
```

```
babynames.csv                    indexes_slicing_sorting.ipynb
```

When we use `pandas` to read in data, we get a DataFrame. A DataFrame is a tabular data structure where each column is labeled (in this case 'Name', 'Sex', 'Count', 'Year') and each row is labeled (in this case 0, 1, 2, ..., 1891893). The Table introduced in Data 8, however, only has labeled columns.

The labels of a DataFrame are called the *indexes* of the DataFrame and make many data manipulations easier.

Indexes, Slicing, and Sorting¶

Let's use `pandas` to answer the following question:

What were the five most popular baby names in 2016?

Breaking the Problem Down¶

We can decompose this question into the following simpler table manipulations:

1. Slice out the rows for the year 2016.
2. Sort the rows in descending order by Count.

Now, we can express these steps in `pandas`.

Slicing using `.loc` ¶

To select subsets of a DataFrame, we use the `.loc` slicing syntax. The first argument is the label of the row and the second is the label of the column:

```
baby
```

	Name	Sex	Count	Year
0	Mary	F	9217	1884
1	Anna	F	3860	1884
2	Emma	F	2587	1884
...	...	...	...	...
1891891	Verna	M	5	1883
1891892	Winnie	M	5	1883
1891893	Winthrop	M	5	1883

1891894 rows × 4 columns

```
baby.loc[1, 'Name'] # Row labeled 1, Column labeled 'Name'
```

```
'Anna'
```

To slice out multiple rows or columns, we can use `:`. Note that `.loc` slicing is inclusive, unlike Python's slicing.

```
# Get rows 1 through 5, columns Name through Count inclusive
baby.loc[1:5, 'Name':'Count']
```

	Name	Sex	Count
1	Anna	F	3860
2	Emma	F	2587
3	Elizabeth	F	2549
4	Minnie	F	2243
5	Margaret	F	2142

We will often want a single column from a DataFrame:

```
baby.loc[:, 'Year']
```

```
0          1884
1          1884
2          1884
           ...
1891891    1883
1891892    1883
1891893    1883
Name: Year, Length: 1891894, dtype: int64
```

Note that when we select a single column, we get a `pandas` Series. A Series is like a one-dimensional NumPy array since we can perform arithmetic on all the elements at once.

```
baby.loc[:, 'Year'] * 2
```

```
0          3768
1          3768
2          3768
           ...
1891891    3766
1891892    3766
1891893    3766
Name: Year, Length: 1891894, dtype: int64
```

To select out specific columns, we can pass a list into the `.loc` slice:

```
# This is a DataFrame again
baby.loc[:, ['Name', 'Year']]
```

	Name	Year
0	Mary	1884
1	Anna	1884
2	Emma	1884
...	...	...
1891891	Verna	1883
1891892	Winnie	1883
1891893	Winthrop	1883

1891894 rows × 2 columns

Selecting columns is common, so there's a shorthand.

```
# Shorthand for baby.loc[:, 'Name']
baby['Name']
```

```
0              Mary
1              Anna
2              Emma
             ...
1891891       Verna
1891892      Winnie
1891893    Winthrop
Name: Name, Length: 1891894, dtype: object
```

```
# Shorthand for baby.loc[:, ['Name', 'Count']]
baby[['Name', 'Count']]
```

	Name	Count
0	Mary	9217
1	Anna	3860
2	Emma	2587
...	...	...
1891891	Verna	5
1891892	Winnie	5
1891893	Winthrop	5

1891894 rows × 2 columns

Slicing rows using a predicate¶

To slice out the rows with year 2016, we will first create a Series containing `True` for each row we want to keep and `False` for each row we want to drop. This is simple because math and boolean operators on Series are applied to each element in the Series.

```
# Series of years
baby['Year']
```

```
0          1884
1          1884
2          1884
           ...
1891891    1883
1891892    1883
1891893    1883
Name: Year, Length: 1891894, dtype: int64
```

```
# Compare each year with 2016
baby['Year'] == 2016
```

```
0          False
1          False
2          False
           ...
1891891    False
1891892    False
1891893    False
Name: Year, Length: 1891894, dtype: bool
```

Once we have this Series of `True` and `False`, we can pass it into `.loc`.

```
# We are slicing rows, so the boolean Series goes in the first
# argument to .loc
baby_2016 = baby.loc[baby['Year'] == 2016, :]
baby_2016
```

	Name	Sex	Count	Year
1850880	Emma	F	19414	2016
1850881	Olivia	F	19246	2016
1850882	Ava	F	16237	2016
...	...	...	...	...
1883745	Zyahir	M	5	2016
1883746	Zyel	M	5	2016
1883747	Zylyn	M	5	2016

32868 rows × 4 columns

Sorting Rows¶

The next step is the sort the rows in descending order by 'Count'. We can use the `sort_values()` function.

```
sorted_2016 = baby_2016.sort_values('Count', ascending=False)
sorted_2016
```

	Name	Sex	Count	Year
1850880	Emma	F	19414	2016
1850881	Olivia	F	19246	2016
1869637	Noah	M	19015	2016
...	...	...	...	...
1868752	Mikaelyn	F	5	2016
1868751	Miette	F	5	2016
1883747	Zylyn	M	5	2016

32868 rows × 4 columns

Finally, we will use `.iloc` to slice out the first five rows of the DataFrame. `.iloc` works like `.loc` but takes in numerical indices instead of labels. It does not include the right endpoint in its slices, like Python's list slicing.

```
# Get the value in the zeroth row, zeroth column
sorted_2016.iloc[0, 0]
```

```
'Emma'
```

```
# Get the first five rows
sorted_2016.iloc[0:5]
```

	Name	Sex	Count	Year
1850880	Emma	F	19414	2016
1850881	Olivia	F	19246	2016
1869637	Noah	M	19015	2016
1869638	Liam	M	18138	2016
1850882	Ava	F	16237	2016

In Conclusion¶

We now have the five most popular baby names in 2016 and learned to express the following operations in `pandas`:

Operation	`pandas`
Read a CSV file	`pd.read_csv()`
Slicing using labels or indices	`.loc` and `.iloc`
Slicing rows using a predicate	Use a boolean-valued Series in `.loc`
Sorting rows	`.sort_values()`

Show Widgets Open on DataHub

Table of Contents

Grouping and Pivoting¶

In this section, we will answer the question:

What were the most popular male and female names in each year?

Here's the Baby Names dataset once again:

```
baby = pd.read_csv('babynames.csv')
baby.head()
# the .head() method outputs the first five rows of the
DataFrame
```

	Name	Sex	Count	Year
0	Mary	F	9217	1884
1	Anna	F	3860	1884
2	Emma	F	2587	1884
3	Elizabeth	F	2549	1884
4	Minnie	F	2243	1884

Breaking the Problem Down¶

We should first notice that the question in the previous section has similarities to this one; the question in the previous section restricts names to babies born in 2016 whereas this question asks for names in all years.

We once again decompose this problem into simpler table manipulations.

1. Group the `baby` DataFrame by 'Year' and 'Sex'.
2. For each group, compute the most popular name.

Recognizing which operation is needed for each problem is sometimes tricky. Usually, a convoluted series of steps will signal to you that there might be a simpler way to express what you want. If we didn't immediately recognize that we needed to group, for example, we might write steps like the following:

1. Loop through each unique year.
2. For each year, loop through each unique sex.
3. For each unique year and sex, find the most common name.

There is almost always a better alternative to looping over a `pandas` DataFrame. **In particular, looping over unique values of a DataFrame should usually be replaced with a group.**

Grouping¶

To group in `pandas`. we use the `.groupby()` method.

```
baby.groupby('Year')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x1a14e21f60>
```

`.groupby()` returns a strange-looking `DataFrameGroupBy` object. We can call `.agg()` on this object with an aggregation function in order to get a familiar output:

```
# The aggregation function takes in a series of values for each
group
# and outputs a single value
def length(series):
    return len(series)

# Count up number of values for each year. This is equivalent to
# counting the number of rows where each year appears.
baby.groupby('Year').agg(length)
```

	Name	Sex	Count
Year			
1880	2000	2000	2000
1881	1935	1935	1935
1882	2127	2127	2127
...	...	...	...
2014	33206	33206	33206
2015	33063	33063	33063
2016	32868	32868	32868

137 rows × 3 columns

You might notice that the `length` function simply calls the `len` function, so we can simplify the code above.

```
baby.groupby('Year').agg(len)
```

	Name	Sex	Count
Year			
1880	2000	2000	2000
1881	1935	1935	1935
1882	2127	2127	2127
...	...	...	...
2014	33206	33206	33206
2015	33063	33063	33063
2016	32868	32868	32868

137 rows × 3 columns

The aggregation is applied to each column of the DataFrame, producing redundant information. We can restrict the output columns by slicing before grouping.

```
year_rows = baby[['Year', 'Count']].groupby('Year').agg(len)
year_rows

# A further shorthand to accomplish the same result:
#
# year_counts = baby[['Year', 'Count']].groupby('Year').count()
#
# pandas has shorthands for common aggregation functions,
including
# count, sum, and mean.
```

	Count
Year	
1880	2000
1881	1935
1882	2127
...	...
2014	33206
2015	33063
2016	32868

137 rows × 1 columns

Note that the index of the resulting DataFrame now contains the unique years, so we can slice subsets of years using `.loc` as before:

```
# Every twentieth year starting at 1880
year_rows.loc[1880:2016:20, :]
```

	Count
Year	
1880	2000
1900	3730
1920	10755
1940	8961
1960	11924
1980	19440
2000	29764

Grouping on Multiple Columns¶

As we've seen in Data 8, we can group on multiple columns to get groups based on unique pairs of values. To do this, pass in a list of column labels into `.groupby()`.

```
grouped_counts = baby.groupby(['Year', 'Sex']).sum()
grouped_counts
```

		Count
Year	**Sex**	
1880	**F**	90992
	M	110491
1881	**F**	91953
...	**...**	...
2015	**M**	1907211
2016	**F**	1756647
	M	1880674

274 rows × 1 columns

The code above computes the total number of babies born for each year and sex. Let's now use grouping by muliple columns to compute the most popular names for each year and sex. Since the data are already sorted in descending order of Count for each year and sex, we can define an aggregation function that returns the first value in each series. (If the data weren't sorted, we can call `sort_values()` first.)

```
# The most popular name is simply the first one that appears in
the series
def most_popular(series):
    return series.iloc[0]

baby_pop = baby.groupby(['Year', 'Sex']).agg(most_popular)
baby_pop
```

		Name	Count
Year	**Sex**		
1880	**F**	Mary	7065
	M	John	9655
1881	**F**	Mary	6919
...	**...**	...	...
2015	**M**	Noah	19594
2016	**F**	Emma	19414
	M	Noah	19015

274 rows × 2 columns

Notice that grouping by multiple columns results in multiple labels for each row. This is called a "multilevel index" and is tricky to work with. The important thing to know is that `.loc` takes in a tuple for the row index instead of a single value:

```
baby_pop.loc[(2000, 'F'), 'Name']
```

```
'Emily'
```

But `.iloc` behaves the same as usual since it uses indices instead of labels:

```
baby_pop.iloc[10:15, :]
```

		Name	Count
Year	**Sex**		
1885	**F**	Mary	9128
	M	John	8756
1886	**F**	Mary	9889
	M	John	9026
1887	**F**	Mary	9888

Pivoting¶

If you group by two columns, you can often use pivot to present your data in a more convenient format. Using a pivot lets you use one set of grouped labels as the columns of the resulting table.

To pivot, use the `pd.pivot_table()` function.

```
pd.pivot_table(baby,
               index='Year',          # Index for rows
               columns='Sex',         # Columns
               values='Name',         # Values in table
               aggfunc=most_popular) # Aggregation function
```

Sex	F	M
Year		
1880	Mary	John
1881	Mary	John
1882	Mary	John
...	...	...
2014	Emma	Noah
2015	Emma	Noah
2016	Emma	Noah

137 rows × 2 columns

Compare this result to the `baby_pop` table that we computed using `.groupby()`. We can see that the `Sex` index in `baby_pop` became the columns of the pivot table.

```
baby_pop
```

		Name	Count
Year	**Sex**		
1880	**F**	Mary	7065
	M	John	9655
1881	**F**	Mary	6919
...	**...**	...	...
2015	**M**	Noah	19594
2016	**F**	Emma	19414
	M	Noah	19015

274 rows × 2 columns

In Conclusion¶

We now have the most popular baby names for each sex and year in our dataset and learned to express the following operations in `pandas`:

Operation	`pandas`
Group	`df.groupby(label)`
Group by multiple columns	`df.groupby([label1, label2])`
Group and aggregate	`df.groupby(label).agg(func)`
Pivot	`pd.pivot_table()`

Show Widgets Open on DataHub

Table of Contents

Apply, Strings, and Plotting¶

In this section, we will answer the question:

Can we use the last letter of a name to predict the sex of the baby?

Here's the Baby Names dataset once again:

```
baby = pd.read_csv('babynames.csv')
baby.head()
# the .head() method outputs the first five rows of the
DataFrame
```

	Name	Sex	Count	Year
0	Mary	F	9217	1884
1	Anna	F	3860	1884
2	Emma	F	2587	1884
3	Elizabeth	F	2549	1884
4	Minnie	F	2243	1884

Breaking the Problem Down¶

Although there are many ways to see whether prediction is possible, we will use plotting in this section. We can decompose this question into two steps:

1. Compute the last letter of each name.

2. Group by the last letter and sex, aggregating on Count.
3. Plot the counts for each sex and letter.

Apply¶

`pandas` Series contain an `.apply()` method that takes in a function and applies it to each value in the Series.

```
names = baby['Name']
names.apply(len)
```

```
0          4
1          4
2          4
          ..
1891891    5
1891892    6
1891893    8
Name: Name, Length: 1891894, dtype: int64
```

To extract the last letter of each name, we can define our own function to pass into `.apply()`:

```
def last_letter(string):
    return string[-1]

names.apply(last_letter)
```

```
0          y
1          a
2          a
          ..
1891891    a
1891892    e
1891893    p
Name: Name, Length: 1891894, dtype: object
```

String Manipulation¶

Although `.apply()` is flexible, it is often faster to use the built-in string manipulation functions in `pandas` when dealing with text data.

`pandas` provides access to string manipulation functions using the `.str` attribute of Series.

```
names = baby['Name']
names.str.len()
```

```
0          4
1          4
2          4
          ..
1891891    5
1891892    6
1891893    8
Name: Name, Length: 1891894, dtype: int64
```

We can directly slice out the last letter of each name in a similar way.

```
names.str[-1]
```

```
0          y
1          a
2          a
          ..
1891891    a
1891892    e
1891893    p
Name: Name, Length: 1891894, dtype: object
```

We suggest looking at the docs for the full list of string methods (link).

We can now add this column of last letters to our `baby` DataFrame.

```
baby['Last'] = names.str[-1]
baby
```

	Name	Sex	Count	Year	Last
0	Mary	F	9217	1884	y
1	Anna	F	3860	1884	a
2	Emma	F	2587	1884	a
...	...	...	...	...	...
1891891	Verna	M	5	1883	a
1891892	Winnie	M	5	1883	e
1891893	Winthrop	M	5	1883	p

1891894 rows × 5 columns

Grouping¶

To compute the sex distribution for each last letter, we need to group by both Last and Sex.

```
# Shorthand for baby.groupby(['Last', 'Sex']).agg(np.sum)
baby.groupby(['Last', 'Sex']).sum()
```

		Count	Year
Last	**Sex**		
a	**F**	58079486	915565667
	M	1931630	53566324
b	**F**	17376	1092953
...	**...**	...	...
y	**M**	18569388	114394474
z	**F**	142023	4268028
	M	120123	9649274

52 rows × 2 columns

Notice that `Year` is also summed up since each non-grouped column is passed into the aggregation function. To avoid this, we can select out the desired columns before calling `.groupby()` .

```
# When lines get long, you can wrap the entire expression in
parentheses
# and insert newlines before each method call
letter_dist = (
    baby[['Last', 'Sex', 'Count']]
    .groupby(['Last', 'Sex'])
    .sum()
)
letter_dist
```

		Count
Last	**Sex**	
a	**F**	58079486
	M	1931630
b	**F**	17376
...	**...**	...
y	**M**	18569388
z	**F**	142023
	M	120123

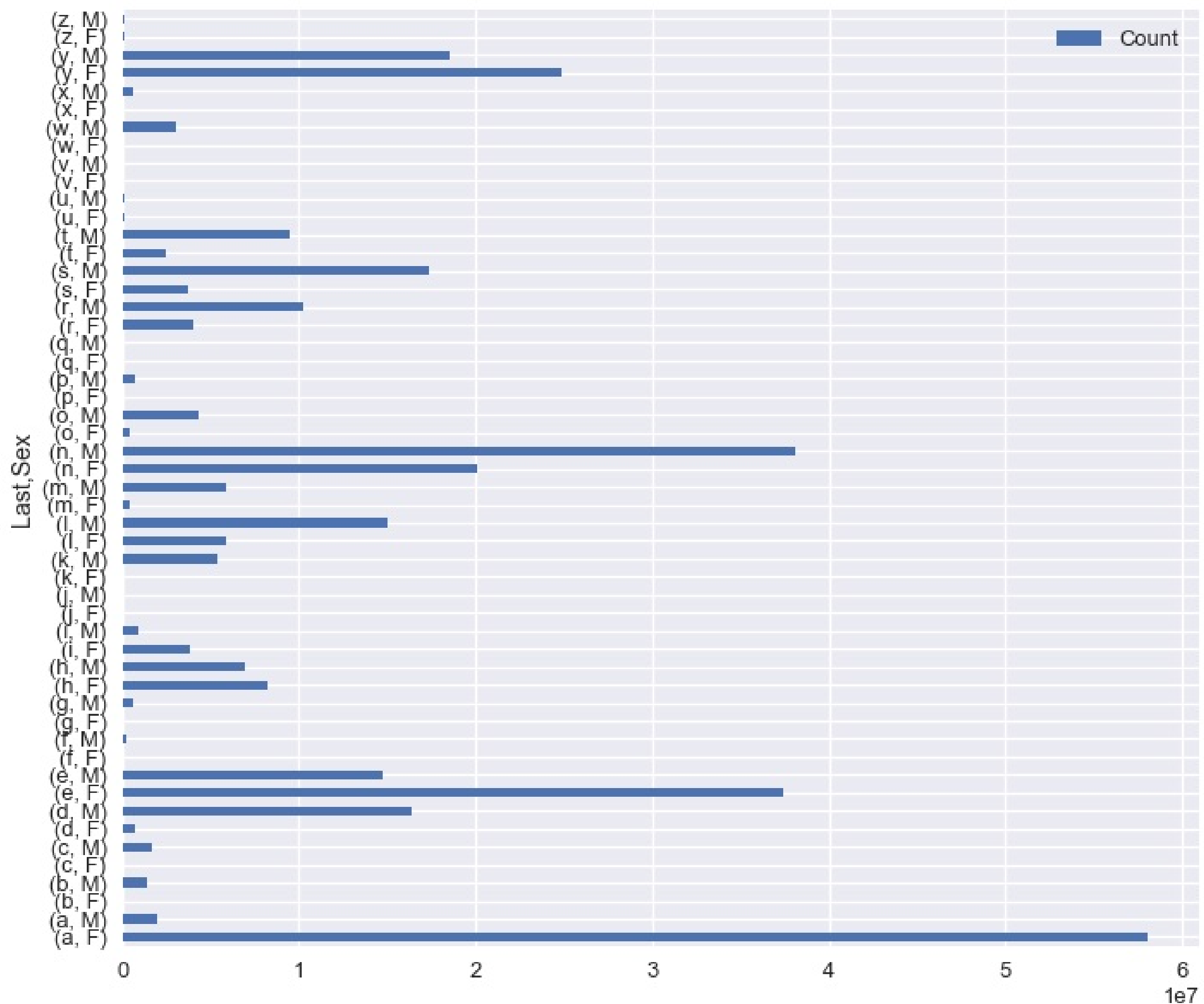
52 rows × 1 columns

Plotting¶

`pandas` provides built-in plotting functionality for most basic plots, including bar charts, histograms, line charts, and scatterplots. To make a plot from a DataFrame, use the `.plot` attribute:

```
# We use the figsize option to make the plot larger
letter_dist.plot.barh(figsize=(10, 10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a17af4780>
```



Although this plot shows the distribution of letters and sexes, the male and female bars are difficult to tell apart. By looking at the `pandas` docs on plotting (link) we learn that `pandas` plots one group of bars for row column in the DataFrame, showing one differently colored bar for each column. This means that a pivoted version of the `letter_dist` table will have the right format.

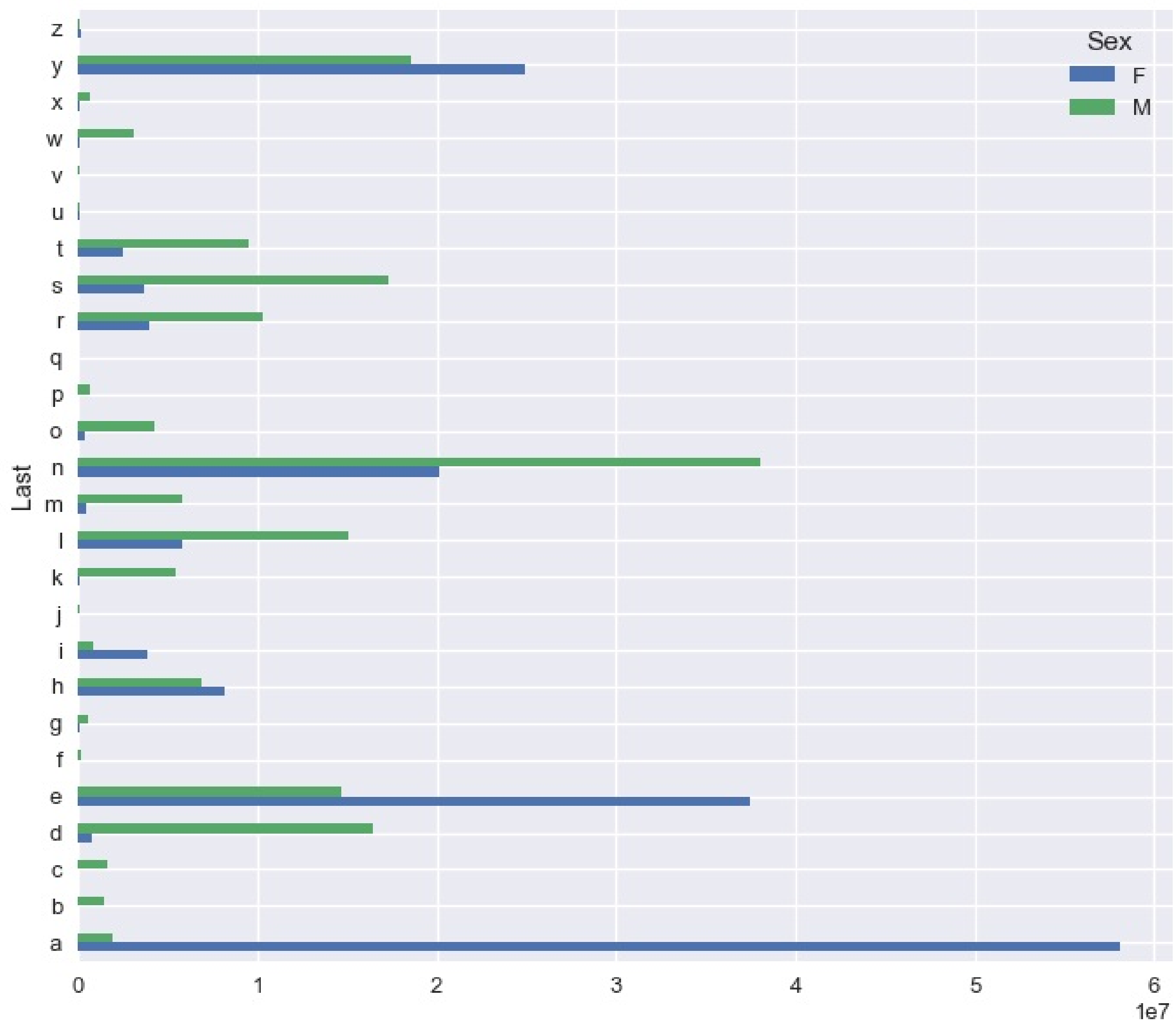
```
letter_pivot = pd.pivot_table(
    baby, index='Last', columns='Sex', values='Count',
aggfunc='sum'
)
letter_pivot
```

Sex	F	M
Last		
a	58079486	1931630
b	17376	1435939
c	30262	1672407
...	...	...
x	37381	644092
y	24877638	18569388
z	142023	120123

26 rows × 2 columns

```
letter_pivot.plot.barh(figsize=(10, 10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a17c36978>
```



Notice that `pandas` conveniently generates a legend for us as well. However, this is still difficult to interpret. We plot the counts for each letter and sex which causes some bars to appear very long and others to be almost invisible. We should instead plot the proportion of male and female babies within each last letter.

```
total_for_each_letter = letter_pivot['F'] + letter_pivot['M']

letter_pivot['F prop'] = letter_pivot['F'] /
total_for_each_letter
letter_pivot['M prop'] = letter_pivot['M'] /
total_for_each_letter
letter_pivot
```

Sex	F	M	F prop	M prop
Last				
a	58079486	1931630	0.967812	0.032188
b	17376	1435939	0.011956	0.988044
c	30262	1672407	0.017773	0.982227
...	...	...	...	...
x	37381	644092	0.054853	0.945147
y	24877638	18569388	0.572597	0.427403
z	142023	120123	0.541771	0.458229

26 rows × 4 columns

```
(letter_pivot[['F prop', 'M prop']]
 .sort_values('M prop') # Sorting orders the plotted bars
 .plot.barh(figsize=(10, 10))
)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a18194b70>
```



In Conclusion¶

We can see that almost all first names that end in 'p' are male and names that end in 'a' are female! In general, the difference between bar lengths for many letters implies that we can often make a good guess to a person's sex if we just know the last letter of their first name.

We've learned to express the following operations in `pandas` :

Operation	`pandas`
Applying a function elementwise	`series.apply(func)`
String manipulation	`series.str.func()`
Plotting	`df.plot.func()`

Data Cleaning

Data come in many formats and vary greatly in usefulness for analysis. Although we would prefer all our data to come in a tabular format with each value recorded consistently and accurately, in reality we must carefully check our data for potential issues that can eventually result in incorrect conclusions.

The term "data cleaning" refers to the process of combing through the data and deciding how to resolve inconsistencies and missing values. We will discuss common problems found in datasets and approaches to address them.

Data cleaning has inherent limitations. For example, no amount of data cleaning will fix a biased sampling process. Before embarking on the sometimes lengthy process of data cleaning, we must be confident that our data are collected accurately and with as little bias as possible. Only then can we investigate the data itself and use data cleaning to resolve issues in the data format or entry process.

We will introduce data cleaning techniques by working with City of Berkeley Police Department datasets.

Show Widgets Open on DataHub

Table of Contents

Investigating Berkeley Police Data¶

We will use the Berkeley Police Department's publicly available datasets to demonstrate data cleaning techniques. We have downloaded the Calls for Service dataset and Stops dataset.

We can use the `ls` shell command with the `-lh` flags to see more details about the files:

```
!ls -lh data/
```

```
total 13936
-rw-r--r--@ 1 sam  staff   979K Aug 29 14:41 Berkeley_PD_-
_Calls_for_Service.csv
-rw-r--r--@ 1 sam  staff    81B Aug 29 14:28 cvdow.csv
-rw-r--r--@ 1 sam  staff   5.8M Aug 29 14:41 stops.json
```

The command above shows the data files and their file sizes. This is especially useful because we now know the files are small enough to load into memory. As a rule of thumb, it is usually safe to load a file into memory that is around one fourth of the total memory capacity of the computer. For example, if a computer has 4GB of RAM we should be able to load a 1GB CSV file in `pandas`. To handle larger datasets we will need additional computational tools that we will cover later in this book.

Notice the use of the exclamation point before `ls`. This tells Jupyter that the next line of code is a shell command, not a Python expression. We can run any available shell command in Jupyter using `!`:

```
# The `wc` shell command shows us how many lines each file has.
# We can see that the `stops.json` file has the most lines
(29852).
!wc -l data/*
```

```
   16497 data/Berkeley_PD_-_Calls_for_Service.csv
       8 data/cvdow.csv
   29852 data/stops.json
   46357 total
```

Understanding the Data Generation¶

We will state important questions you should ask of all datasets before data cleaning or processing. These questions are related to how the data were generated, so data cleaning will usually **not** be able to resolve issues that arise here.

What do the data contain? The website for the Calls for Service data states that the dataset describes "crime incidents (not criminal reports) within the last 180 days". Further reading reveals that "not all calls for police service are included (e.g. Animal Bite)".

The website for the Stops data states that the dataset contains data on all "vehicle detentions (including bicycles) and pedestrian detentions (up to five persons)" since January 26, 2015.

Are the data a census? This depends on our population of interest. For example, if we are interested in calls for service within the last 180 days for crime incidents then the Calls dataset is a census. However, if we are interested in calls for service within the last 10 years the dataset is clearly not a census. We can make similar statements about the Stops dataset since the data collection started on January 26, 2015.

If the data form a sample, is it a probability sample? If we are investigating a period of time that the data do not have entries for, the data do not form a probability sample since there is no randomness involved in the data collection process — we have all data for certain time periods but no data for others.

What limitations will this data have on our conclusions? Although we will ask this question at each step of our data processing, we can already see that our data impose important limitations. The most important limitation is that we cannot make unbiased estimations for time periods not covered by our datasets.

Cleaning The Calls Dataset¶

Let's now clean the Calls dataset. The `head` shell command prints the first five lines of the file.

```
!head data/Berkeley_PD_-_Calls_for_Service.csv
```

```
CASENO,OFFENSE,EVENTDT,EVENTTM,CVLEGEND,CVDOW,InDbDate,Block_Loc
ation,BLKADDR,City,State
17091420,BURGLARY AUTO,07/23/2017 12:00:00 AM,06:00,BURGLARY -
VEHICLE,0,08/29/2017 08:28:05 AM,"2500 LE CONTE AVE
Berkeley, CA
(37.876965, -122.260544)",2500 LE CONTE AVE,Berkeley,CA
17020462,THEFT FROM PERSON,04/13/2017 12:00:00
AM,08:45,LARCENY,4,08/29/2017 08:28:00 AM,"2200 SHATTUCK AVE
Berkeley, CA
(37.869363, -122.268028)",2200 SHATTUCK AVE,Berkeley,CA
17050275,BURGLARY AUTO,08/24/2017 12:00:00 AM,18:30,BURGLARY -
VEHICLE,4,08/29/2017 08:28:06 AM,"200 UNIVERSITY AVE
Berkeley, CA
(37.865491, -122.310065)",200 UNIVERSITY AVE,Berkeley,CA
```

It appears to be a comma-separated values (CSV) file, though it's hard to tell whether the entire file is formatted properly. We can use `pd.read_csv` to read in the file as a DataFrame. If `pd.read_csv` errors, we will have to dig deeper and manually resolve formatting issues. Fortunately, `pd.read_csv` successfully returns a DataFrame:

```
calls = pd.read_csv('data/Berkeley_PD_-_Calls_for_Service.csv')
calls
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	Block_Loc
0	17091420	BURGLARY AUTO	07/23/2017 12:00:00 AM	06:00	...	2500 LE CC AVE\nBerke CA\n(37.876 -...
1	17020462	THEFT FROM PERSON	04/13/2017 12:00:00 AM	08:45	...	2200 SHATTUCK AVE\nBerke CA\n(37.869 -...
2	17050275	BURGLARY AUTO	08/24/2017 12:00:00 AM	18:30	...	200 UNIVERSIT AVE\nBerke CA\n(37.865 ...
...	...	...	...	...	...	...
5505	17018126	DISTURBANCE	04/01/2017 12:00:00 AM	12:22	...	1600 FAIRV ST\nBerkele CA\n(37.850 -1...
5506	17090665	THEFT MISD. (UNDER $950)	04/01/2017 12:00:00 AM	12:00	...	2000 DELAWARE ST\nBerkele CA\n(37.874 -1...
5507	17049700	SEXUAL ASSAULT MISD.	08/22/2017 12:00:00 AM	20:02	...	2400 TELEGRAP AVE\nBerke CA\n(37.866 ...

5508 rows × 11 columns

We can define a function to show different slices of the data and then interact with it:

```
def df_interact(df):
    '''
    Outputs sliders that show rows and columns of df
    '''
    def peek(row=0, col=0):
        return df.iloc[row:row + 5, col:col + 6]
    interact(peek, row=(0, len(df), 5), col=(0, len(df.columns)
- 6))
    print('({} rows, {} columns) total'.format(df.shape[0],
df.shape[1]))

df_interact(calls)
```

Show Widget

```
(5508 rows, 11 columns) total
```

Based on the output above, the resulting DataFrame looks reasonably well-formed since the columns are properly named and the data in each column seems to be entered consistently. What data does each column contain? We can look at the dataset website:

Column	Description	Type
CASENO	Case Number	Number
OFFENSE	Offense Type	Plain Text
EVENTDT	Date Event Occurred	Date & Time
EVENTTM	Time Event Occurred	Plain Text
CVLEGEND	Description of Event	Plain Text
CVDOW	Day of Week Event Occurred	Number
InDbDate	Date dataset was updated in the portal	Date & Time
Block_Location	Block level address of event	Location
BLKADDR		Plain Text
City		Plain Text
State		Plain Text

On the surface the data looks easy to work with. However, before starting data analysis we must answer the following questions:

1. **Are there missing values in the dataset?** This question is important because missing

values can represent many different things. For example, missing addresses could mean that locations were removed to protect anonymity, or that some respondents chose not to answer a survey question, or that a recording device broke.

2. **Are there any missing values that were filled in (e.g. a 999 for unknown age or 12:00am for unknown date)?** These will clearly impact analysis if we ignore them.
3. **Which parts of the data were entered by a human?** As we will soon see, human-entered data is filled with inconsistencies and mispellings.

Although there are plenty more checks to go through, these three will suffice for many cases. See the Quartz bad data guide for a more complete list of checks.

Are there missing values?¶

This is a simple check in `pandas`:

```
# True if row contains at least one null value
null_rows = calls.isnull().any(axis=1)
calls[null_rows]
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	Block_Locat
116	17014831	BURGLARY AUTO	03/16/2017 12:00:00 AM	22:00	...	Berkeley, CA\n(37.869(-122.270455)
478	17042511	BURGLARY AUTO	07/20/2017 12:00:00 AM	16:00	...	Berkeley, CA\n(37.869(-122.270455)
486	17022572	VEHICLE STOLEN	04/22/2017 12:00:00 AM	21:00	...	Berkeley, CA\n(37.869(-122.270455)
...	...	...	...	...	...	...
4945	17091287	VANDALISM	07/01/2017 12:00:00 AM	08:00	...	Berkeley, CA\n(37.869(-122.270455)
4947	17038382	BURGLARY RESIDENTIAL	06/30/2017 12:00:00 AM	15:00	...	Berkeley, CA\n(37.869(-122.270455)
5167	17091632	VANDALISM	08/15/2017 12:00:00 AM	23:30	...	Berkeley, CA\n(37.869(-122.270455)

27 rows × 11 columns

It looks like 27 calls didn't have a recorded address in BLKADDR. Unfortunately, the data description isn't very clear on how the locations were recorded. We know that all of these calls were made for events in Berkeley, so we can likely assume that the addresses for these calls were originally somewhere in Berkeley.

Are there any missing values that were filled in?¶

From the missing value check above we can see that the Block_Location column has Berkeley, CA recorded if the location was missing.

In addition, an inspection of the `calls` table shows us that the EVENTDT column has the correct dates but records 12am for all of its times. Instead, the times are in the EVENTTM column.

```
# Show the first 7 rows of the table again for reference
calls.head(7)
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	Block_Locatior
0	17091420	BURGLARY AUTO	07/23/2017 12:00:00 AM	06:00	...	2500 LE CONTE AVE\nBerkeley, CA\n(37.876965 -...
1	17020462	THEFT FROM PERSON	04/13/2017 12:00:00 AM	08:45	...	2200 SHATTUCK AVE\nBerkeley, CA\n(37.869363 -...
2	17050275	BURGLARY AUTO	08/24/2017 12:00:00 AM	18:30	...	200 UNIVERSITY AVE\nBerkeley, CA\n(37.865491 ...
3	17019145	GUN/WEAPON	04/06/2017 12:00:00 AM	17:30	...	1900 SEVENTH ST\nBerkeley, CA\n(37.869318 -12...
4	17044993	VEHICLE STOLEN	08/01/2017 12:00:00 AM	18:00	...	100 PARKSIDE DR\nBerkeley, CA\n(37.854247 -12...
5	17037319	BURGLARY RESIDENTIAL	06/28/2017 12:00:00 AM	12:00	...	1500 PRINCE ST\nBerkeley, CA\n(37.851503 -122...
6	17030791	BURGLARY RESIDENTIAL	05/30/2017 12:00:00 AM	08:45	...	300 MENLO PL\nBerkeley, CA\n

7 rows × 11 columns

As a data cleaning step, we want to merge the EVENTDT and EVENTTM columns to record both date and time in one field. If we define a function that takes in a DF and returns a new DF, we can later use `pd.pipe` to apply all transformations in one go.

```
def combine_event_datetimes(calls):
    combined = pd.to_datetime(
        # Combine date and time strings
        calls['EVENTDT'].str[:10] + ' ' + calls['EVENTTM'],
        infer_datetime_format=True,
    )
    return calls.assign(EVENTDTTM=combined)

# To peek at the result without mutating the calls DF:
calls.pipe(combine_event_datetimes).head(2)
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	BLKADDR	
0	17091420	BURGLARY AUTO	07/23/2017 12:00:00 AM	06:00	...	2500 LE CONTE AVE	Berl
1	17020462	THEFT FROM PERSON	04/13/2017 12:00:00 AM	08:45	...	2200 SHATTUCK AVE	Berl

2 rows × 12 columns

Which parts of the data were entered by a human?¶

It looks like most of the data columns are machine-recorded, including the date, time, day of week, and location of the event.

In addition, the OFFENSE and CVLEGEND columns appear to contain consistent values. We can check the unique values in each column to see if anything was misspelled:

```
calls['OFFENSE'].unique()
```

```
array(['BURGLARY AUTO', 'THEFT FROM PERSON', 'GUN/WEAPON',
       'VEHICLE STOLEN', 'BURGLARY RESIDENTIAL', 'VANDALISM',
       'DISTURBANCE', 'THEFT MISD. (UNDER $950)', 'THEFT FROM
AUTO',
       'DOMESTIC VIOLENCE', 'THEFT FELONY (OVER $950)', 'ALCOHOL
OFFENSE',
       'MISSING JUVENILE', 'ROBBERY', 'IDENTITY THEFT',
       'ASSAULT/BATTERY MISD.', '2ND RESPONSE', 'BRANDISHING',
       'MISSING ADULT', 'NARCOTICS', 'FRAUD/FORGERY',
       'ASSAULT/BATTERY FEL.', 'BURGLARY COMMERCIAL', 'MUNICIPAL
CODE',
       'ARSON', 'SEXUAL ASSAULT FEL.', 'VEHICLE RECOVERED',
       'SEXUAL ASSAULT MISD.', 'KIDNAPPING', 'VICE',
'HOMICIDE'], dtype=object)
```

```
calls['CVLEGEND'].unique()
```

```
array(['BURGLARY - VEHICLE', 'LARCENY', 'WEAPONS OFFENSE',
       'MOTOR VEHICLE THEFT', 'BURGLARY - RESIDENTIAL',
'VANDALISM',
       'DISORDERLY CONDUCT', 'LARCENY - FROM VEHICLE', 'FAMILY
OFFENSE',
       'LIQUOR LAW VIOLATION', 'MISSING PERSON', 'ROBBERY',
'FRAUD',
       'ASSAULT', 'NOISE VIOLATION', 'DRUG VIOLATION',
       'BURGLARY - COMMERCIAL', 'ALL OTHER OFFENSES', 'ARSON',
'SEX CRIME',
       'RECOVERED VEHICLE', 'KIDNAPPING', 'HOMICIDE'],
dtype=object)
```

Since each value in these columns appears to be spelled correctly, we won't have to perform any corrections on these columns.

We also check the BLKADDR column for inconsistencies and find that sometimes an address is recorded (e.g. 2500 LE CONTE AVE) but other times a cross street is recorded (e.g. ALLSTON WAY & FIFTH ST). This suggests that a human entered this data in and this column will be difficult to use for analysis. Fortunately we can use the latitude and longitude of the event instead of the street address.

```
calls['BLKADDR'][[0, 5001]]
```

```
0              2500 LE CONTE AVE
5001     ALLSTON WAY & FIFTH ST
Name: BLKADDR, dtype: object
```

Final Touchups¶

This dataset seems almost ready for analysis. The Block_Location column seems to contain strings that record address, latitude, and longitude. We will want to separate the latitude and longitude for easier use.

```
def split_lat_lon(calls):
    return calls.join(
        calls['Block_Location']
        # Get coords from string
        .str.split('\n').str[2]
        # Remove parens from coords
        .str[1:-1]
        # Split latitude and longitude
        .str.split(', ', expand=True)
        .rename(columns={0: 'Latitude', 1: 'Longitude'})
    )

calls.pipe(split_lat_lon).head(2)
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	City	State
0	17091420	BURGLARY AUTO	07/23/2017 12:00:00 AM	06:00	...	Berkeley	CA
1	17020462	THEFT FROM PERSON	04/13/2017 12:00:00 AM	08:45	...	Berkeley	CA

2 rows × 13 columns

Then, we can match the day of week number with its weekday:

```
# This DF contains the day for each number in CVDOW
day_of_week = pd.read_csv('data/cvdow.csv')
day_of_week
```

	CVDOW	Day
0	0	Sunday
1	1	Monday
2	2	Tuesday
3	3	Wednesday
4	4	Thursday
5	5	Friday
6	6	Saturday

```
def match_weekday(calls):
    return calls.merge(day_of_week, on='CVDOW')
calls.pipe(match_weekday).head(2)
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	BLKADDR	
0	17091420	BURGLARY AUTO	07/23/2017 12:00:00 AM	06:00	...	2500 LE CONTE AVE	Ber
1	17038302	BURGLARY AUTO	07/02/2017 12:00:00 AM	22:00	...	BOWDITCH STREET & CHANNING WAY	Ber

2 rows × 12 columns

We'll drop columns we no longer need:

```
def drop_unneeded_cols(calls):
    return calls.drop(columns=['CVDOW', 'InDbDate',
'Block_Location', 'City',
                               'State', 'EVENTDT', 'EVENTTM'])
```

Finally, we'll pipe the `calls` DF through all the functions we've defined:

```
calls_final = (calls.pipe(combine_event_datetimes)
               .pipe(split_lat_lon)
               .pipe(match_weekday)
               .pipe(drop_unneeded_cols))
df_interact(calls_final)
```

Show Widget

```
(5508 rows, 8 columns) total
```

The Calls dataset is now ready for further data analysis. In the next section, we will clean the Stops dataset.

Show Widgets Open on DataHub

Table of Contents

Cleaning The Stops Dataset¶

The Stops dataset (webpage) records police stops of pedestrians and vehicles. Let's prepare it for further analysis.

We can use the `head` command to display the first few lines of the file.

```
!head data/stops.json
```

```
{
  "meta" : {
    "view" : {
      "id" : "6e9j-pj9p",
      "name" : "Berkeley PD - Stop Data",
      "attribution" : "Berkeley Police Department",
      "averageRating" : 0,
      "category" : "Public Safety",
      "createdAt" : 1444171604,
      "description" : "This data was extracted from the
Department’s Public Safety Server and covers the data beginning
January 26, 2015.  On January 26, 2015 the department began
collecting data pursuant to General Order B-4 (issued December
31, 2014).  Under that order, officers were required to provide
certain data after making all vehicle detentions (including
bicycles) and pedestrian detentions (up to five persons).  This
data set lists stops by police in the categories of traffic,
suspicious vehicle, pedestrian and bicycle stops.  Incident
number, date and time, location and disposition codes are also
listed in this data.\r\n\r\nAddress data has been changed from a
specific address, where applicable, and listed as the block
where the incident occurred.  Disposition codes were entered by
officers who made the stop.  These codes included the person(s)
race, gender, age (range), reason for the stop, enforcement
action taken, and whether or not a search was
conducted.\r\n\r\nThe officers of the Berkeley Police Department
are prohibited from biased based policing, which is defined as
any police-initiated action that relies on the race, ethnicity,
or national origin rather than the behavior of an individual or
information that leads the police to a particular individual who
has been identified as being engaged in criminal activity.",
```


The `stops.json` file is clearly not a CSV file. In this case, the file contains data in the JSON (JavaScript Object Notation) format, a commonly used data format where data is recorded in a dictionary format. Python's `json` module makes reading in this file as a dictionary simple.

```
import json

# Note that this could cause our computer to run out of memory
if the file
# is large. In this case, we've verified that the file is small
enough to
# read in beforehand.
with open('data/stops.json') as f:
    stops_dict = json.load(f)

stops_dict.keys()
```

```
dict_keys(['meta', 'data'])
```

Note that `stops_dict` is a Python dictionary, so displaying it will display the entire dataset in the notebook. This could cause the browser to crash, so we only display the keys of the dictionary above. To peek at the data without potentially crashing the browser, we can print the dictionary to a string and only output some of the first characters of the string.

```
from pprint import pformat

def print_dict(dictionary, num_chars=1000):
    print(pformat(dictionary)[:num_chars])

print_dict(stops_dict['meta'])
```

```
{'view': {'attribution': 'Berkeley Police Department',
          'averageRating': 0,
          'category': 'Public Safety',
          'columns': [{'dataTypeName': 'meta_data',
                       'fieldName': ':sid',
                       'flags': ['hidden'],
                       'format': {},
                       'id': -1,
                       'name': 'sid',
                       'position': 0,
                       'renderTypeName': 'meta_data'},
                      {'dataTypeName': 'meta_data',
                       'fieldName': ':id',
                       'flags': ['hidden'],
                       'format': {},
                       'id': -1,
                       'name': 'id',
                       'position': 0,
                       'renderTypeName': 'meta_data'},
                      {'dataTypeName': 'meta_data',
                       'fieldName': ':position',
                       'flags': ['hidden'],
                       'format': {},
```


```
print_dict(stops_dict['data'], num_chars=300)
```

```
[[1,
  '29A1B912-A0A9-4431-ADC9-FB375809C32E',
  1,
  1444146408,
  '932858',
  1444146408,
  '932858',
  None,
  '2015-00004825',
  '2015-01-26T00:10:00',
  'SAN PABLO AVE / MARIN AVE',
  'T',
  'M',
  None,
  None],
 [2,
  '1644D161-1113-4C4F-BB2E-BF780E7AE73E',
  2,
  1444146408,
  '932858',
  14
```

We can likely deduce that the `'meta'` key in the dictionary contains a description of the data and its columns and the `'data'` contains a list of data rows. We can use this information to initialize a DataFrame.

```
# Load the data from JSON and assign column titles
stops = pd.DataFrame(
    stops_dict['data'],
    columns=[c['name'] for c in stops_dict['meta']['view']
['columns']])

stops
```

	sid	id	position	created_at	...	Incident Type	D
0	1	29A1B912-A0A9-4431-ADC9-FB375809C32E	1	1444146408	...	T	M
1	2	1644D161-1113-4C4F-BB2E-BF780E7AE73E	2	1444146408	...	T	M
2	3	5338ABAB-1C96-488D-B55F-6A47AC505872	3	1444146408	...	T	M
...	...	...	...	...	...	...	..
29205	31079	C2B606ED-7872-4B0B-BC9B-4EF45149F34B	31079	1496269085	...	T	B
29206	31080	8FADF18D-7FE9-441D-8709-7BFEABDACA7A	31080	1496269085	...	T	H
29207	31081	F60BD2A4-8C47-4BE7-B1C6-4934BE9DF838	31081	1496269085	...	1194	A

29208 rows × 15 columns

```
# Prints column names
stops.columns
```

```
Index(['sid', 'id', 'position', 'created_at', 'created_meta',
'updated_at',
       'updated_meta', 'meta', 'Incident Number', 'Call
Date/Time', 'Location',
       'Incident Type', 'Dispositions', 'Location - Latitude',
       'Location - Longitude'],
      dtype='object')
```

The website contains documentation about the following columns:

Column	Description	Type
Incident Number	Number of incident created by Computer Aided Dispatch (CAD) program	Plain Text
Call Date/Time	Date and time of the incident/stop	Date & Time
Location	General location of the incident/stop	Plain Text
Incident Type	This is the occurred incident type created in the CAD program. A code signifies a traffic stop (T), suspicious vehicle stop (1196), pedestrian stop (1194) and bicycle stop (1194B).	Plain Text
Dispositions	Ordered in the following sequence: 1st Character = Race, as follows: A (Asian) B (Black) H (Hispanic) O (Other) W (White) 2nd Character = Gender, as follows: F (Female) M (Male) 3rd Character = Age Range, as follows: 1 (Less than 18) 2 (18-29) 3 (30-39), 4 (Greater than 40) 4th Character = Reason, as follows: I (Investigation) T (Traffic) R (Reasonable Suspicion) K (Probation/Parole) W (Wanted) 5th Character = Enforcement, as follows: A (Arrest) C (Citation) O (Other) W (Warning) 6th Character = Car Search, as follows: S (Search) N (No Search) Additional dispositions may also appear. They are: P - Primary case report M - MDT narrative only AR - Arrest report only (no case report submitted) IN - Incident report FC - Field Card CO - Collision investigation report MH - Emergency Psychiatric Evaluation TOW - Impounded vehicle 0 or 00000 – Officer made a stop of more than five persons	Plain Text
Location - Latitude	General latitude of the call. This data is only uploaded after January 2017	Number
Location - Longitude	General longitude of the call. This data is only uploaded after January 2017.	Number

Notice that the website doesn't contain descriptions for the first 8 columns of the `stops` table. Since these columns appear to contain metadata that we're not interested in analyzing this time, we drop them from the table.

```
columns_to_drop = ['sid', 'id', 'position', 'created_at',
'created_meta',
                   'updated_at', 'updated_meta', 'meta']

# This function takes in a DF and returns a DF so we can use it
for .pipe
def drop_unneeded_cols(stops):
    return stops.drop(columns=columns_to_drop)

stops.pipe(drop_unneeded_cols)
```

	Incident Number	Call Date/Time	Location	Incident Type	Dispositions	
0	2015-00004825	2015-01-26T00:10:00	SAN PABLO AVE / MARIN AVE	T	M	Nc
1	2015-00004829	2015-01-26T00:50:00	SAN PABLO AVE / CHANNING WAY	T	M	Nc
2	2015-00004831	2015-01-26T01:03:00	UNIVERSITY AVE / NINTH ST	T	M	Nc
...	...	...	...	...	...	...
29205	2017-00024245	2017-04-30T22:59:26	UNIVERSITY AVE/6TH ST	T	BM2TWN;	Nc
29206	2017-00024250	2017-04-30T23:19:27	UNIVERSITY AVE / WEST ST	T	HM4TCS;	37
29207	2017-00024254	2017-04-30T23:38:34	CHANNING WAY / BOWDITCH ST	1194	AR;	37

29208 rows × 7 columns

As with the Calls dataset, we will answer the following three questions about the Stops dataset:

1. Are there missing values in the dataset?
2. Are there any missing values that were filled in (e.g. a 999 for unknown age or 12:00am for unknown date)?

3. Which parts of the data were entered by a human?

Are there missing values?¶

We can clearly see that there are many missing latitude and longitudes. The data description states that these two columns are only filled in after Jan 2017.

```
# True if row contains at least one null value
null_rows = stops.isnull().any(axis=1)

stops[null_rows]
```

	Incident Number	Call Date/Time	Location	Incident Type	Dispositions	Lo La
0	2015-00004825	2015-01-26T00:10:00	SAN PABLO AVE / MARIN AVE	T	M	Nc
1	2015-00004829	2015-01-26T00:50:00	SAN PABLO AVE / CHANNING WAY	T	M	Nc
2	2015-00004831	2015-01-26T01:03:00	UNIVERSITY AVE / NINTH ST	T	M	Nc
...	...	...	...	...	...	...
29078	2017-00023764	2017-04-29T01:59:36	2180 M L KING JR WAY	1194	BM4IWN;	Nc
29180	2017-00024132	2017-04-30T12:54:23	6TH/UNI	1194	M;	Nc
29205	2017-00024245	2017-04-30T22:59:26	UNIVERSITY AVE/6TH ST	T	BM2TWN;	Nc

25067 rows × 7 columns

We can check the other columns for missing values:

```
# True if row contains at least one null value without checking
# the latitude and longitude columns
null_rows = stops.iloc[:, :-2].isnull().any(axis=1)

df_interact(stops[null_rows])
```

Show Widget

```
(63 rows, 7 columns) total
```

By browsing through the table above, we can see that all other missing values are in the Dispositions column. Unfortunately, we do not know from the data description why these Dispositions might be missing. Since only there are only 63 missing values compared to 25,000 rows in the original table, we can proceed with analysis while being mindful that these missing values could impact results.

Are there any missing values that were filled in?¶

It doesn't seem like any previously missing values were filled in for us. Unlike in the Calls dataset where the date and time were in separate columns, the Call Date/Time column in the Stops dataset contains both date and time.

Which parts of the data were entered by a human?¶

As with the Calls dataset, it looks like most of the columns in this dataset were recorded by a machine or were a category selected by a human (e.g. Incident Type).

However, the Location column doesn't have consistently entered values. Sure enough, we spot some typos in the data:

```
stops['Location'].value_counts()
```

```
2200 BLOCK SHATTUCK AVE             229
37.8693028530001~-122.272234021     213
UNIVERSITY AVE / SAN PABLO AVE      202
                                   ...
VALLEY ST / DWIGHT WAY                1
COLLEGE AVE / SIXTY-THIRD ST          1
GRIZZLY PEAK BLVD / MARIN AVE         1
Name: Location, Length: 6393, dtype: int64
```

What a mess! It looks like sometimes an address was entered, sometimes a cross-street, and other times a latitude-longitude pair. Unfortunately, we don't have very complete latitude-longitude data to use in place of this column. We may have to manually clean this column if we want to use locations for future analysis.

We can also check the Dispositions column:

```
dispositions = stops['Dispositions'].value_counts()

# Outputs a slider to pan through the unique Dispositions in
# order of how often they appear
interact(lambda row=0: dispositions.iloc[row:row+7],
         row=(0, len(dispositions), 7))
```

Show Widget

```
<function __main__.<lambda>>
```

The Dispositions columns also contains inconsistencies. For example, some dispositions start with a space, some end with a semicolon, and some contain multiple entries. The variety of values suggests that this field contains human-entered values and should be treated with caution.

```
# Strange values...
dispositions.iloc[[0, 20, 30, 266, 1027]]
```

```
M            1683
M;            238
 M            176
HF4TWN;        14
 OM4KWS         1
Name: Dispositions, dtype: int64
```

In addition, the most common disposition is `M` which isn't a permitted first character in the Dispositions column. This could mean that the format of the column changed over time or that officers are allowed to enter in the disposition without matching the format in the data description. In any case, the column will be challenging to work with.

We can take some simple steps to clean the Dispositions column by removing leading and trailing whitespace, removing trailing semi-colons, and replacing the remaining semi-colons with commas.

```
def clean_dispositions(stops):
    cleaned = (stops['Dispositions']
               .str.strip()
               .str.rstrip(';')
               .str.replace(';', ','))
    return stops.assign(Dispositions=cleaned)
```

As before, we can now pipe the `stops` DF through the cleaning functions we've defined:

```
stops_final = (stops
               .pipe(drop_unneeded_cols)
               .pipe(clean_dispositions))
df_interact(stops_final)
```

Show Widget

```
(29208 rows, 7 columns) total
```

Conclusion¶

As these two datasets have shown, data cleaning can often be both difficult and tedious. Cleaning 100% of the data often takes too long, but not cleaning the data at all results in faulty conclusions; we have to weigh our options and strike a balance each time we encounter a new dataset.

The decisions made during data cleaning impact all future analyses. For example, we chose not to clean the Location column of the Stops dataset so we should treat that column with caution. Each decision made during data cleaning should be carefully documented for future reference, preferably in a notebook so that both code and explanations appear together.

Exporatory Data Analysis

> Exploratory data analysis is an attitude, a state of flexibility, a willingness to look for those things that we believe are not there, as well as those we believe to be there.
>
> — John Tukey

In Exploratory Data Analysis (EDA), the third step of the data science lifecycle, we summarize, visualize, and transform the data in order to understand it more deeply. In particular, through EDA we identify potential issues in the data and discover trends that inform further analyses.

We seek to understand the following properties about our data:

1. Structure: the format of our data file.
2. Granularity: how fine or coarse each row and column is.
3. Scope: how (in)complete our data are.
4. Temporality: how the data are situation in time.
5. Faithfulness: how well the data captures "reality".

Although we introduce data cleaning and EDA separately to help organize this book, in practice you will often switch between the two. For example, visualizing a column may show misformatted values that you should use data cleaning techniques to process. With this in mind, we return to the Berkeley Police Department datasets for exploration.

Show Widgets Open on DataHub

Table of Contents

Structure¶

The structure of a dataset refers to the "shape" of the data files. At a basic level, this refers to the format that the data are entered in. For example, we saw that the Calls dataset is a comma-separated values file:

```
!head data/Berkeley_PD_-_Calls_for_Service.csv
```

```
CASENO,OFFENSE,EVENTDT,EVENTTM,CVLEGEND,CVDOW,InDbDate,Block_Loc
ation,BLKADDR,City,State
17091420,BURGLARY AUTO,07/23/2017 12:00:00 AM,06:00,BURGLARY -
VEHICLE,0,08/29/2017 08:28:05 AM,"2500 LE CONTE AVE
Berkeley, CA
(37.876965, -122.260544)",2500 LE CONTE AVE,Berkeley,CA
17020462,THEFT FROM PERSON,04/13/2017 12:00:00
AM,08:45,LARCENY,4,08/29/2017 08:28:00 AM,"2200 SHATTUCK AVE
Berkeley, CA
(37.869363, -122.268028)",2200 SHATTUCK AVE,Berkeley,CA
17050275,BURGLARY AUTO,08/24/2017 12:00:00 AM,18:30,BURGLARY -
VEHICLE,4,08/29/2017 08:28:06 AM,"200 UNIVERSITY AVE
Berkeley, CA
(37.865491, -122.310065)",200 UNIVERSITY AVE,Berkeley,CA
```

The Stops dataset, on the other hand, is a JSON (JavaScript Object Notation) file.

```
# Show first and last 5 lines of file
!head -n 5 data/stops.json
!echo '...'
!tail -n 5 data/stops.json
```

```
{
  "meta" : {
    "view" : {
      "id" : "6e9j-pj9p",
      "name" : "Berkeley PD - Stop Data",
...
, [ 31079, "C2B606ED-7872-4B0B-BC9B-4EF45149F34B", 31079,
1496269085, "932858", 1496269085, "932858", null, "2017-
00024245", "2017-04-30T22:59:26", " UNIVERSITY AVE/6TH ST", "T",
"BM2TWN; ", null, null ]
, [ 31080, "8FADF18D-7FE9-441D-8709-7BFEABDACA7A", 31080,
1496269085, "932858", 1496269085, "932858", null, "2017-
00024250", "2017-04-30T23:19:27", " UNIVERSITY AVE /  WEST ST",
"T", "HM4TCS; ", "37.8698757000001", "-122.286550846" ]
, [ 31081, "F60BD2A4-8C47-4BE7-B1C6-4934BE9DF838", 31081,
1496269085, "932858", 1496269085, "932858", null, "2017-
00024254", "2017-04-30T23:38:34", " CHANNING WAY /  BOWDITCH
ST", "1194", "AR; ", "37.867207539", "-122.256529377" ]
 ]
}
```

Of course, there are many other types of data formats. Here is a list of the most common formats:

- Comma-Separated Values (CSV) and Tab-Separated Values (TSV). These files contain tabular data delimited by either a comma for CSV or a tab character (`\t`) for TSV. These files are typically easy to work with because the data are entered in a similar format to DataFrames.
- JavaScript Object Notation (JSON). These files contain data in a nested dictionary format. Typically we have to read in the entire file as a Python dict and then figure out how to extract fields for a DataFrame from the dict.
- eXtensible Markup Language (XML) or HyperText Markup Language (HTML). These files also contain data in a nested format, for example:

```
<?xml version="1.0" encoding="UTF-8"?>
  <note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
```

In a later chapter we will use XPath to extract data from these types of files.

- Log data. Many applications will output some data as they run in an unstructured text format, for example:

```
2005-03-23 23:47:11,663 - sa - INFO - creating an instance of aux_module.Aux
2005-03-23 23:47:11,665 - sa.aux.Aux - INFO - creating an instance of Aux
2005-03-23 23:47:11,665 - sa - INFO - created an instance of aux_module.Aux
2005-03-23 23:47:11,668 - sa - INFO - calling aux_module.Aux.do_something
2005-03-23 23:47:11,668 - sa.aux.Aux - INFO - doing something
```

In a later chapter we will use Regular Expressions to extract data from these types of files.

Joins¶

Data will often be split across multiple tables. For example, one table can describe some people's personal information while another will contain their emails:

```
people = pd.DataFrame(
    [["Joey",      "blue",    42,  "M"],
     ["Weiwei",    "blue",    50,  "F"],
     ["Joey",      "green",    8,  "M"],
     ["Karina",    "green",    7,  "F"],
     ["Nhi",       "blue",     3,  "F"],
     ["Sam",       "pink",   -42,  "M"]],
    columns = ["Name", "Color", "Number", "Sex"])

people
```

	Name	Color	Number	Sex
0	Joey	blue	42	M
1	Weiwei	blue	50	F
2	Joey	green	8	M
3	Karina	green	7	F
4	Fernando	pink	-9	M
5	Nhi	blue	3	F
6	Sam	pink	-42	M

```
email = pd.DataFrame(
    [["Deb",  "deborah_nolan@berkeley.edu"],
     ["Sam",  "samlau95@berkeley.edu"],
     ["John", "doe@nope.com"],
     ["Joey", "jegonzal@cs.berkeley.edu"],
     ["Weiwei", "weiwzhang@berkeley.edu"],
     ["Weiwei", "weiwzhang+123@berkeley.edu"],
     ["Karina", "kgoot@berkeley.edu"]],
    columns = ["User Name", "Email"])

email
```

	User Name	Email
0	Deb	deborah_nolan@berkeley.edu
1	Sam	samlau95@berkeley.edu
2	John	doe@nope.com
3	Joey	jegonzal@cs.berkeley.edu
4	Weiwei	weiwzhang@berkeley.edu
5	Weiwei	weiwzhang+123@berkeley.edu
6	Karina	kgoot@berkeley.edu

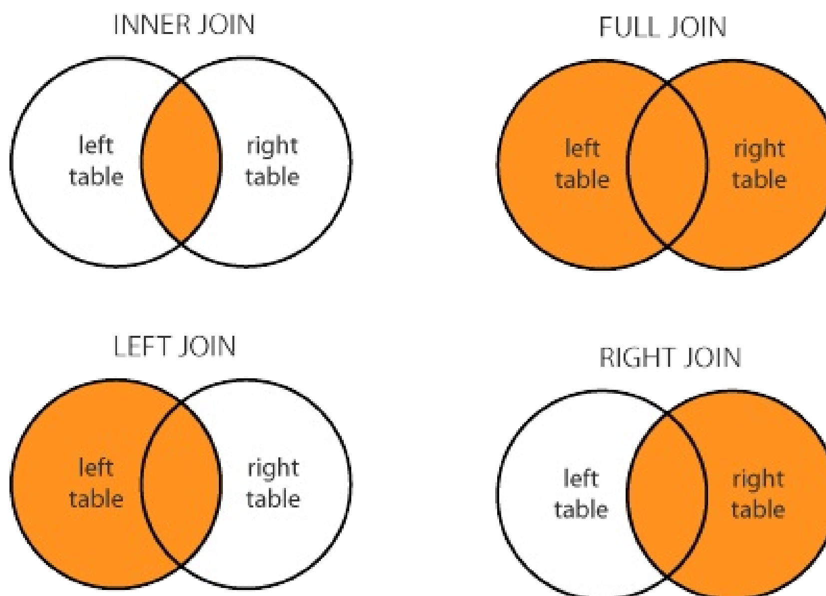
To match up each person with his or her email, we can join the two tables on the columns that contain the usernames. We must then decide what to do about people that appear in one table but not the other. For example, Fernando appears in the `people` table but not the

`email` table. We have several types of joins for each strategy of matching missing values. One of the more common joins is the *inner join*, where any row that doesn't have a match is dropped in the final result:

```
# Fernando, Nhi, Deb, and John don't appear
people.merge(email, how='inner', left_on='Name', right_on='User Name')
```

	Name	Color	Number	Sex	User Name	Email
0	Joey	blue	42	M	Joey	jegonzal@cs.berkeley.edu
1	Joey	green	8	M	Joey	jegonzal@cs.berkeley.edu
2	Weiwei	blue	50	F	Weiwei	weiwzhang@berkeley.edu
3	Weiwei	blue	50	F	Weiwei	weiwzhang+123@berkeley.edu
4	Karina	green	7	F	Karina	kgoot@berkeley.edu
5	Sam	pink	-42	M	Sam	samlau95@berkeley.edu

There are four basic joins that we use most often: inner, full (sometimes called "outer"), left, and right joins. Below is a diagram to show the difference between these types of joins.



Use the dropdown menu below to show the result of the four different types of joins on the `people` and `email` tables. Notice which rows contain NaN values for outer, left, and right joins.

Show Widget

Structure Checklist¶

You should have answers to the following questions after looking at the structure of your datasets. We will answer them for the Calls and Stops datasets.

Are the data in a standard format or encoding?

Standard formats include:

- Tabular data: CSV, TSV, Excel, SQL
- Nested data: JSON, XML

The Calls dataset came in the CSV format while the Stops dataset came in the JSON format.

Are the data organized in records (e.g. rows)? If not, can we define records by parsing the data?

The Calls dataset came in rows; we extracted records from the Stops dataset.

Are the data nested? If so, can we reasonably unnest the data?

The Calls dataset wasn't nested; we didn't have to work too hard to unnest data from the Stops dataset.

Do the data reference other data? If so, can we join the data?

The Calls dataset references the day of week table. Joining those two tables gives us the day of week for each incident in the dataset. The Stops dataset had no obvious references.

What are the fields (e.g. columns) in each record? What is the type of each column?

The fields for the Calls and Stops datasets are described in the Data Cleaning sections for each dataset.

Show Widgets Open on DataHub

Table of Contents

Granularity¶

The granularity of your data is what each record in your data represents. For example, in the Calls dataset each record represents a single case of a police call.

	CASENO	OFFENSE	CVLEGEND	BLKADDR	EVENTDTTM
0	17091420	BURGLARY AUTO	BURGLARY - VEHICLE	2500 LE CONTE AVE	2017-07-23 06:00:00
1	17038302	BURGLARY AUTO	BURGLARY - VEHICLE	BOWDITCH STREET & CHANNING WAY	2017-07-02 22:00:00
2	17049346	THEFT MISD. (UNDER $950)	LARCENY	2900 CHANNING WAY	2017-08-20 23:20:00
3	17091319	THEFT MISD. (UNDER $950)	LARCENY	2100 RUSSELL ST	2017-07-09 04:15:00
4	17044238	DISTURBANCE	DISORDERLY CONDUCT	TELEGRAPH AVENUE & DURANT AVE	2017-07-30 01:16:00

In the Stops dataset, each record represents a single incident of a police stop.

	Incident Number	Call Date/Time	Location	Incident Type	Dispositions	Location - Latitude
0	2015-00004825	2015-01-26 00:10:00	SAN PABLO AVE / MARIN AVE	T	M	NaN
1	2015-00004829	2015-01-26 00:50:00	SAN PABLO AVE / CHANNING WAY	T	M	NaN
2	2015-00004831	2015-01-26 01:03:00	UNIVERSITY AVE / NINTH ST	T	M	NaN
3	2015-00004848	2015-01-26 07:16:00	2000 BLOCK BERKELEY WAY	1194	BM4ICN	NaN
4	2015-00004849	2015-01-26 07:43:00	1700 BLOCK SAN PABLO AVE	1194	BM4ICN	NaN

On the other hand, we could have received the Stops data in the following format:

	Num Incidents
Call Date/Time	
2015-01-26	46
2015-01-27	57
2015-01-28	56
...	...
2017-04-28	82
2017-04-29	86
2017-04-30	59

825 rows × 1 columns

In this case, each record in the table corresponds to a single date instead of a single incident. We would describe this table as having a coarser granularity than the one above. It's important to know the granularity of your data because it determines what kind of analyses you can perform. Generally speaking, too fine of a granularity is better than too coarse; while we can use grouping and pivoting to change a fine granularity to a coarse one, we have few tools to go from coarse to fine.

Granularity Checklist¶

You should have answers to the following questions after looking at the granularity of your datasets. We will answer them for the Calls and Stops datasets.

What does a record represent?

In the Calls dataset, each record represents a single case of a police call. In the Stops dataset, each record represents a single incident of a police stop.

Do all records capture granularity at the same level? (Sometimes a table will contain summary rows.)

Yes, for both Calls and Stops datasets.

If the data were aggregated, how was the aggregation performed? Sampling and averaging are are common aggregations.

No aggregations were performed as far as we can tell for the datasets. We do keep in mind that in both datasets, the location is entered as a block location instead of a specific address.

What kinds of aggregations can we perform on the data?

For example, it's often useful to aggregate individual people to demographic groups or individual events to totals across time.

In this case, we can aggregate across various granularities of date or time. For example, we can find the most common hour of day for incidents with aggregation. We might also be able to aggregate across event locations to find the regions of Berkeley with the most incidents.

Show Widgets Open on DataHub

Table of Contents

Scope¶

The scope of the dataset refers to the coverage of the dataset in relation to what we are interested in analyzing. We seek to answer the following question about our data scope:

Does the data cover the topic of interest?

For example, the Calls and Stops datasets contain call and stop incidents made in Berkeley. If we are interested in crime incidents in the state of California, however, these datasets will be too limited in scope.

In general, larger scope is more useful than smaller scope since we can filter larger scope down to a smaller scope but often can't go from smaller scope to larger scope. For example, if we had a dataset of police stops in the United States we could subset the dataset to investigate Berkeley.

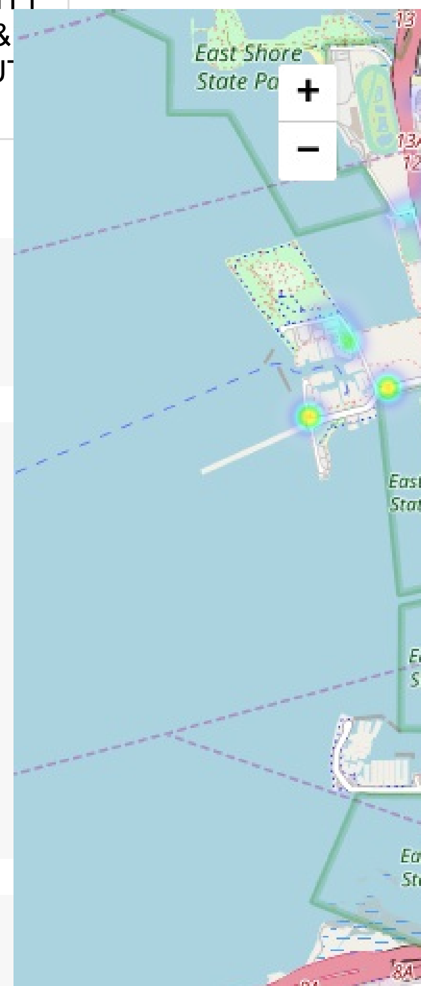
Keep in mind that scope is a broad term not always used to describe geographic location. For example, it can also refer to time coverage — the Calls dataset only contains data for a 180 day period.

We will often address the scope of the dataset during the investigation of the data generation process and confirm the dataset's scope during EDA. Let's confirm the geographic and time scope of the Calls dataset.

```
calls
```

	CASENO	OFFENSE	CVLEGEND	BLKADDR	EVENTDTT
0	17091420	BURGLARY AUTO	BURGLARY - VEHICLE	2500 LE CONTE AVE	2017-07-23 06:00:00
1	17038302	BURGLARY AUTO	BURGLARY - VEHICLE	BOWDITCH STREET & CHANNING WAY	2017-07-02 22:00:00
2	17049346	THEFT MISD. (UNDER $950)	LARCENY	2900 CHANNING WAY	2017-08-20 23:20:00
...	...	...	...	...	...
5505	17021604	IDENTITY THEFT	FRAUD	100 MONTROSE RD	2017-03-31 00:00:00
5506	17033201	DISTURBANCE	DISORDERLY CONDUCT	2300 COLLEGE AVE	2017-06-09 22:34:00
5507	17047247	BURGLARY AUTO	BURGLARY - VEHICLE	UNIVERSITY AVENUE & CHESTNU ST	

5508 rows × 8 columns




```
# Shows earliest and latest dates in calls
calls['EVENTDTTM'].dt.date.sort_values()
```

```
1384    2017-03-02
1264    2017-03-02
1408    2017-03-02
           ...
3516    2017-08-28
3409    2017-08-28
3631    2017-08-28
Name: EVENTDTTM, Length: 5508, dtype: object
```

```
calls['EVENTDTTM'].dt.date.max() -
calls['EVENTDTTM'].dt.date.min()
```

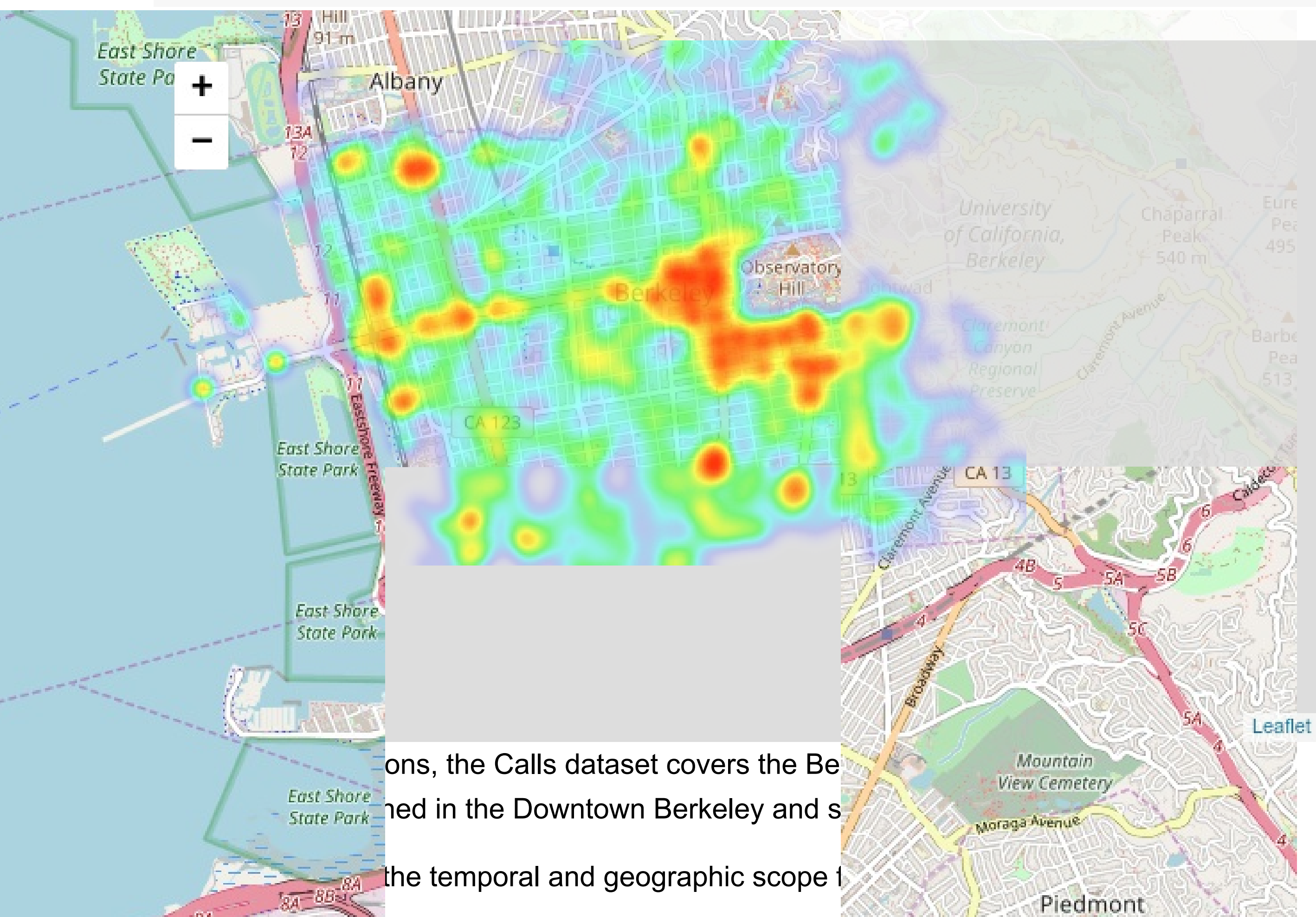
```
datetime.timedelta(179)
```

The table contains data for a time period of 179 days which is close enough to the 180 day time period in the data description that we can suppose there were no calls on either April 14st, 2017 or August 29, 2017.

To check the geographic scope, we can use a map:

```
import folium # Use the Folium Javascript Map Library
import folium.plugins

SF_COORDINATES = (37.87, -122.28)
sf_map = folium.Map(location=SF_COORDINATES, zoom_start=13)
locs = calls[['Latitude',
'Longitude']].astype('float').dropna().as_matrix()
heatmap = folium.plugins.HeatMap(locs.tolist(), radius = 10)
sf_map.add_child(heatmap)
```



ons, the Calls dataset covers the Be

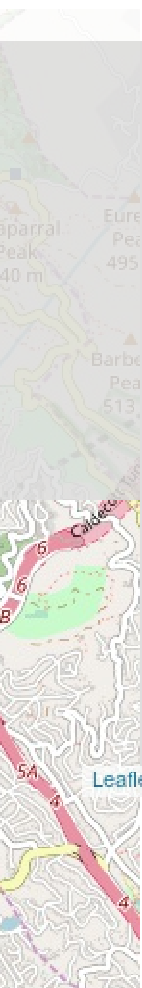
ned in the Downtown Berkeley and s

the temporal and geographic scope t

```
stops
```

	Incident Number	Call Date/Time	Location	Incident Type	Dispositions	Lo - La
0	2015-00004825	2015-01-26 00:10:00	SAN PABLO AVE / MARIN AVE	T	M	NaN
1	2015-00004829	2015-01-26 00:50:00	SAN PABLO AVE / CHANNING WAY	T	M	NaN
2	2015-00004831	2015-01-26 01:03:00	UNIVERSITY AVE / NINTH ST	T	M	
...	...	...	...	...	...	
29205	2017-00024245	2017-04-30 22:59:26	UNIVERSITY AVE/6TH ST	T	BM2	
29206	2017-00024250	2017-04-30 23:19:27	UNIVERSITY AVE / WEST ST	T	HM4	
29207	2017-00024254	2017-04-30 23:38:34	CHANNING WAY / BOWDITCH ST	1194	AR	





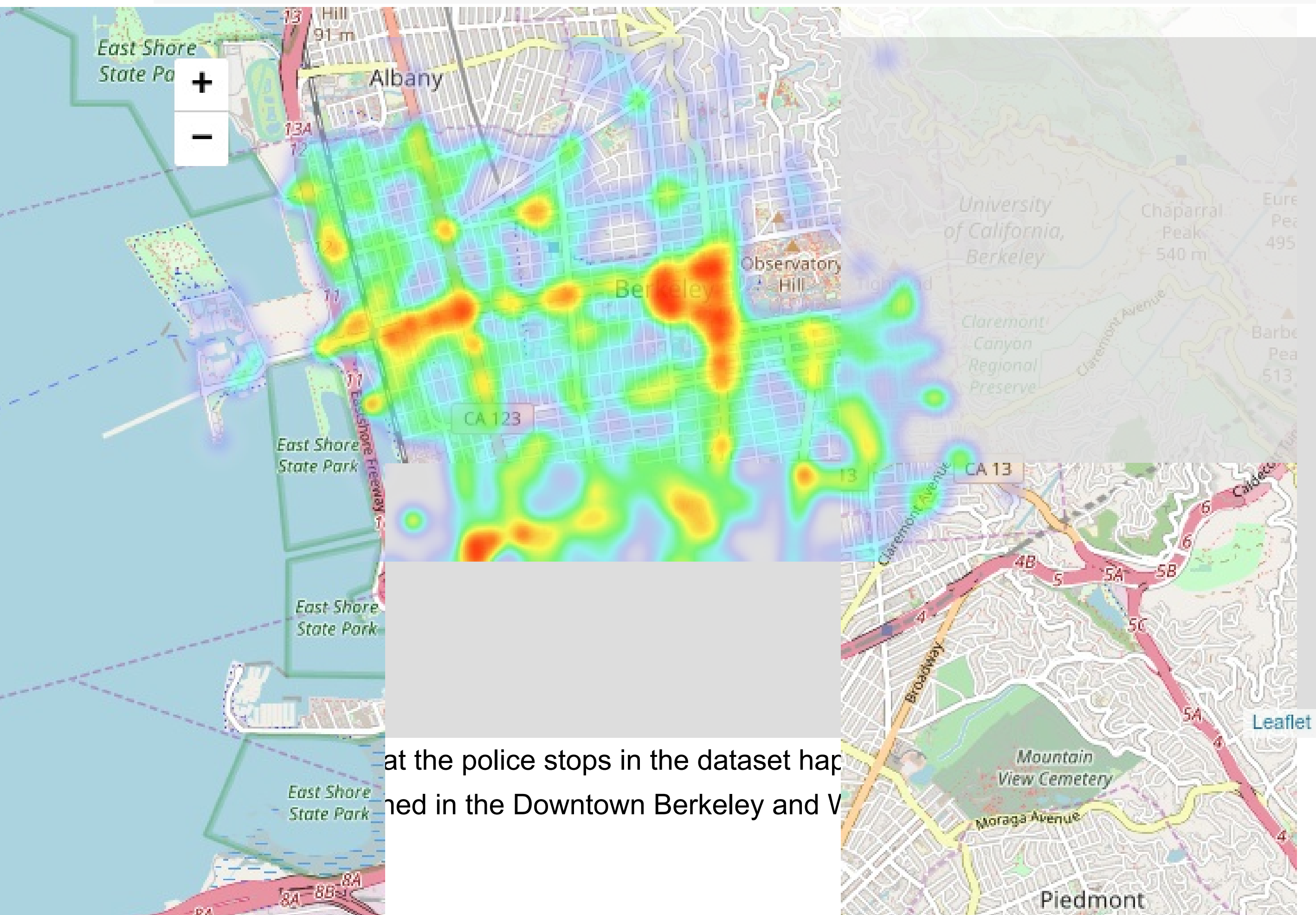
29208 rows × 7 columns

```
stops['Call Date/Time'].dt.date.sort_values()
```

```
0        2015-01-26
25       2015-01-26
26       2015-01-26
            ...
29175    2017-04-30
29177    2017-04-30
29207    2017-04-30
Name: Call Date/Time, Length: 29208, dtype: object
```

As promised, the data collection begins on January 26th, 2015. It looks like the data were downloaded somewhere around the beginning of May 2017 since the dates stop on April 30th, 2017. Let's draw a map to see the geographic data:

```
SF_COORDINATES = (37.87, -122.28)
sf_map = folium.Map(location=SF_COORDINATES, zoom_start=13)
locs = stops[['Location - Latitude', 'Location -
Longitude']].astype('float').dropna().as_matrix()
heatmap = folium.plugins.HeatMap(locs.tolist(), radius = 10)
sf_map.add_child(heatmap)
```



at the police stops in the dataset hap
ned in the Downtown Berkeley and V

Show Widgets Open on DataHub

Table of Contents

Temporality¶

Temporality refers to how the data are situated in time and specifically to the date and time fields in the dataset. We seek to understand the following traits about these fields:

What is the meaning of the date and time fields in the dataset?

In the Calls and Stops dataset, the datetime fields represent when the call or stop was made by the police. However, the Stops dataset also originally had a datetime field recording when the case was entered into the database which we took out during data cleaning since we didn't think it would be useful for analysis.

In addition, we should be careful to note the timezone and Daylight Savings for datetime fields especially when dealing with data that comes from multiple locations.

What representation do the date and time fields have in the data?

Although the US uses the MM/DD/YYYY format, many other countries use the DD/MM/YYYY format. There are still more formats in use around the world and it's important to recognize these differences when analyzing data.

In the Calls and Stops dataset, the dates came in the MM/DD/YYYY format.

Are there strange timestamps that might represent null values?

Some programs use placeholder datetimes instead of null values. For example, Excel's default date is Jan 1st, 1990 and on Excel for Mac, it's Jan 1st, 1904. Many applications will generate a default datetime of 12:00am Jan 1st, 1970 or 11:59pm Dec 31st, 1969 since this is the Unix Epoch for timestamps. If you notice multiple instances of these timestamps in your data, you should take caution and double check your data sources. Neither Calls nor Stops dataset contain any of these suspicious values.

Show Widgets Open on DataHub

Table of Contents

Faithfulness¶

We describe a dataset as "faithful" if we believe it accurately captures reality. Typically, untrustworthy datasets contain:

Unrealistic or incorrect values

For example, dates in the future, locations that don't exist, negative counts, or large outliers.

Violations of obvious dependencies

For example, age and birthday for individuals don't match.

Hand-entered data

As we have seen, these are typically filled with spelling errors and inconsistencies.

Clear signs of data falsification

For example, repeated names, fake looking email addresses, or repeated use of uncommon names or fields.

Notice the many similarities to data cleaning. As we have mentioned, we often go back and forth between data cleaning and EDA, especially when determining data faithfulness. For example, visualizations often help us identify strange entries in the data.

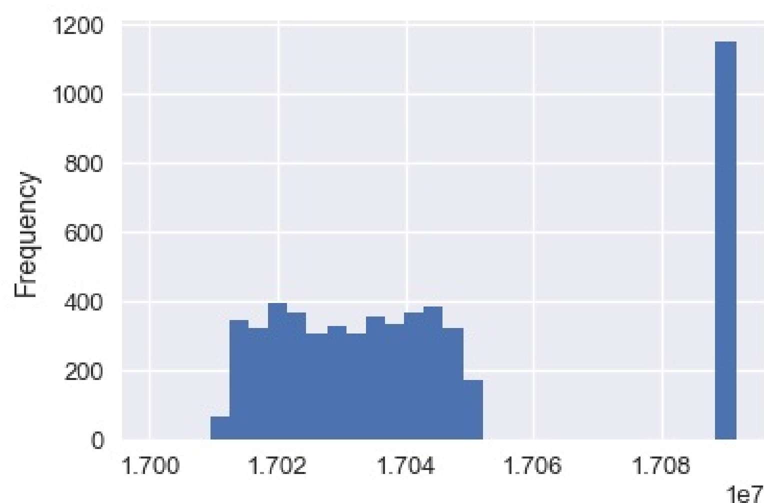
```
calls = pd.read_csv('data/calls.csv')
calls.head()
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	...	BLKADDR
0	17091420	BURGLARY AUTO	07/23/2017 12:00:00 AM	06:00	...	2500 LE CONTE AVE
1	17038302	BURGLARY AUTO	07/02/2017 12:00:00 AM	22:00	...	BOWDITCH STREET & CHANNING WAY
2	17049346	THEFT MISD. (UNDER $950)	08/20/2017 12:00:00 AM	23:20	...	2900 CHANNING WAY
3	17091319	THEFT MISD. (UNDER $950)	07/09/2017 12:00:00 AM	04:15	...	2100 RUSSELL ST
4	17044238	DISTURBANCE	07/30/2017 12:00:00 AM	01:16	...	TELEGRAPH AVENUE & DURANT AVE

5 rows × 9 columns

```
calls['CASENO'].plot.hist(bins=30)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1ebb2898>
```



Notice the unexpected clusters at 17030000 and 17090000. By plotting the distribution of case numbers, we can quickly see anomalies in the data. In this case, we might guess that two different teams of police use different sets of case numbers for their calls.

Exploring the data often reveals anomalies; if fixable, we can then apply data cleaning techniques.

Data Visualization

> There is a magic in graphs. The profile of a curve reveals in a flash a whole situation — the life history of an epidemic, a panic, or an era of prosperity. The curve informs the mind, awakens the imagination, convinces.
>
> — Henry D. Hubbard

Data visualization is an essential tool for data science at every step of analysis, from data cleaning to EDA to communicating conclusions and predictions. Because human minds are highly developed for visual perception, a well-chosen plot can often reveal trends and anomalies in the data much more efficiently than a textual description.

To effectively use data visualizations, you must be proficient with both the programming tools to generate plots and the principles of visualization. In this chapter we will introduce `seaborn` and `matplotlib`, our tools of choice for creating plots. We will also learn how to spot misleading visualizations and how to improve visualizations using data transformations, smoothing, and dimensionality reduction.

Show Widgets Open on DataHub

Table of Contents

Visualizing Quantitative Data¶

We generally use different types of charts to visualize quantitative (numerical) data and qualitative (ordinal or nominal) data.

For quantitative data, we most often use histograms, box plots, and scatter plots.

We can use the seaborn plotting library to create these plots in Python. We will use a dataset containing information about passengers aboard the Titanic.

```
# Import seaborn and apply its plotting styles
import seaborn as sns
sns.set()
```

```
# Load the dataset and drop N/A values to make plot function
calls simpler
ti = sns.load_dataset('titanic').dropna().reset_index(drop=True)

# This table is too large to fit onto a page so we'll output
sliders to
# pan through different sections.
df_interact(ti)
```

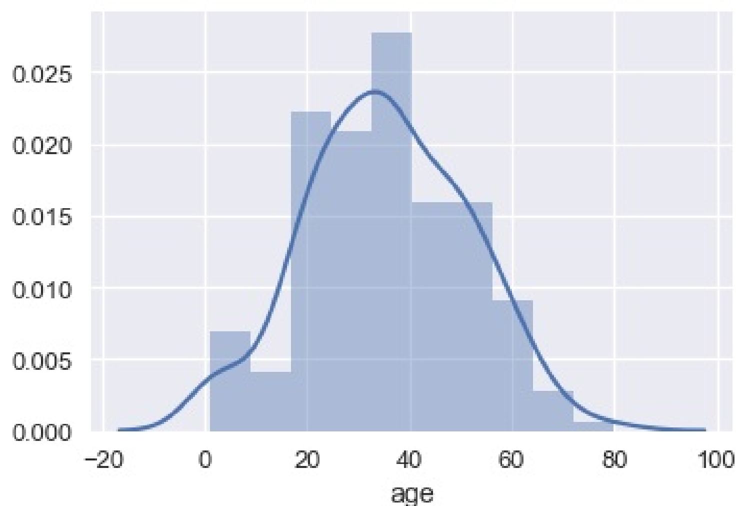
Show Widget

```
(182 rows, 15 columns) total
```

Histograms¶

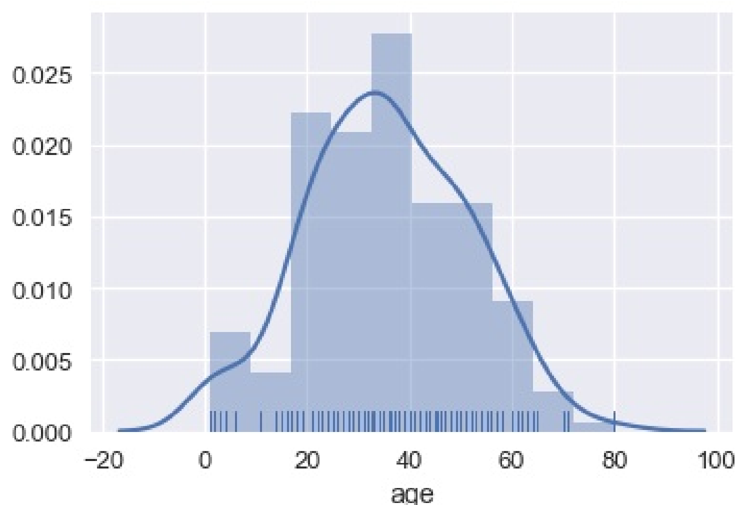
We can see that the dataset contains one row for every passenger. Each row includes the age of the passenger and the amount the passenger paid for a ticket. Let's visualize the ages using a histogram. We can use seaborn's `distplot` function:

```
# Adding a semi-colon at the end tells Jupyter not to output the
# usual <matplotlib.axes._subplots.AxesSubplot> line
sns.distplot(ti['age']);
```



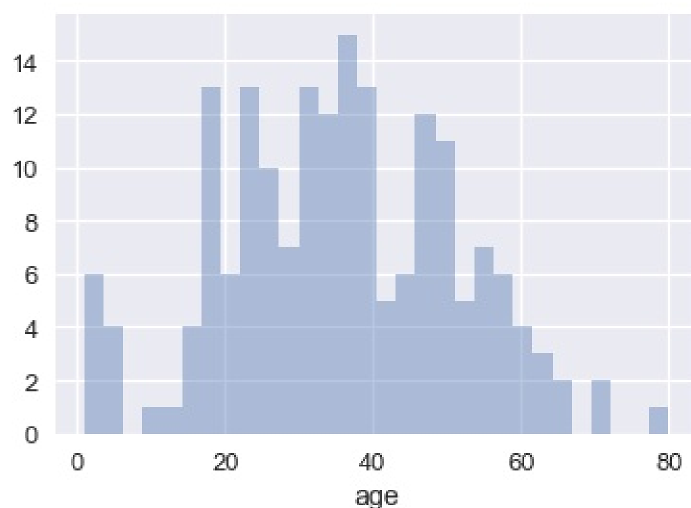
By default, seaborn's `distplot` function will output a smoothed curve that roughly fits the distribution. We can also add a rugplot which marks each individual point on the x-axis:

```
sns.distplot(ti['age'], rug=True);
```



We can also plot the distribution itself. Adjusting the number of bins shows that there were a number of children on board.

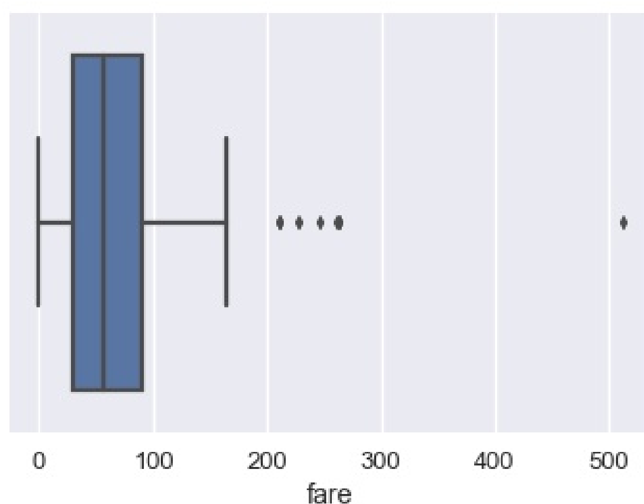
```
sns.distplot(ti['age'], kde=False, bins=30);
```



Box plots¶

Box plots are a convenient way to see where most of the data lie. Typically, we use the 25th and 75th percentiles of the data as the start and endpoints of the box and draw a line within the box for the 50th percentile (the median). We draw two "whiskers" that extend to show the the remaining data except outliers, which are marked as individual points outside the whiskers.

```
sns.boxplot(x='fare', data=ti);
```



We typically use the Inter-Quartile Range (IQR) to determine which points are considered outliers for the box plot. The IQR is the difference between the 75th percentile of the data and the 25th percentile.

```
lower, upper = np.percentile(ti['fare'], [25, 75])
iqr = upper - lower
iqr
```

```
60.29999999999997
```

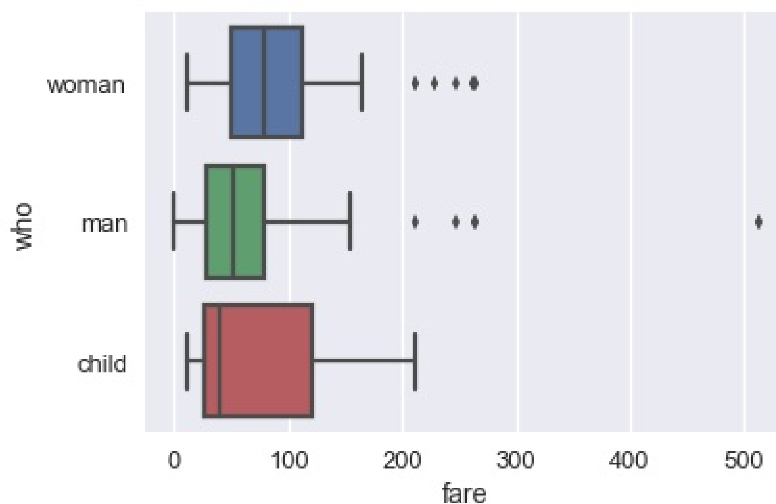
Values greater than 1.5 $\times$ IQR above the 75th percentile and less than 1.5 $\times$ IQR below the 25th percentile are considered outliers and we can see them marked indivdiually on the boxplot above:

```
upper_cutoff = upper + 1.5 * iqr
lower_cutoff = lower - 1.5 * iqr
upper_cutoff, lower_cutoff
```

```
(180.44999999999999, -60.749999999999986)
```

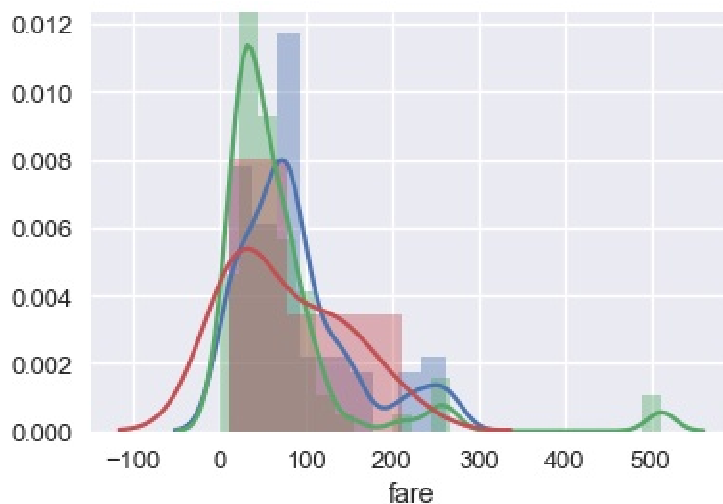
Although histograms show the entire distribution at once, box plots are often easier to understand when we split the data by different categories. For example, we can make one box plot for each passenger type:

```
sns.boxplot(x='fare', y='who', data=ti);
```



The separate box plots are much easier to understand than the overlaid histogram below which plots the same data:

```
sns.distplot(ti.loc[ti['who'] == 'woman', 'fare'])
sns.distplot(ti.loc[ti['who'] == 'man', 'fare'])
sns.distplot(ti.loc[ti['who'] == 'child', 'fare']);
```



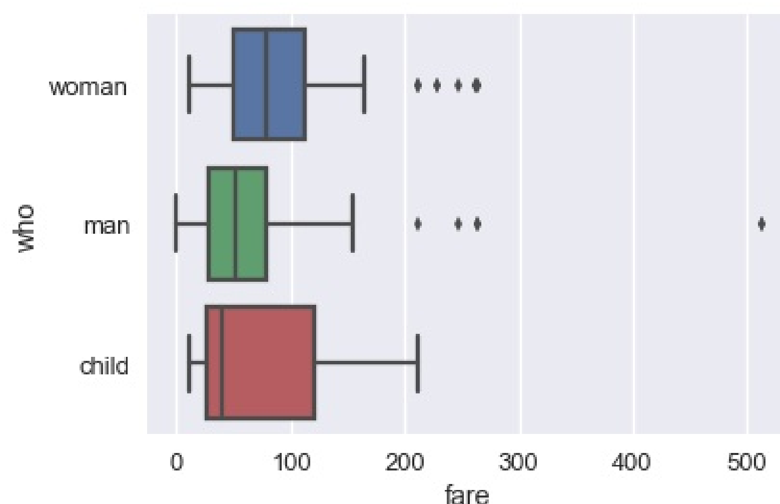
Brief Aside on Using Seaborn¶

You may have noticed that the `boxplot` call to make separate box plots for the `who` column was simpler than the equivalent code to make an overlaid histogram. Although `sns.distplot` takes in an array or Series of data, most other seaborn functions allow you to pass in a DataFrame and specify which column to plot on the x and y axes. For example:

```
# Plots the `fare` column of the `ti` DF on the x-axis
sns.boxplot(x='fare', data=ti);
```

When the column is categorical (the `'who'` column contained `'woman'`, `'man'`, and `'child'`), seaborn will automatically split the data by category before plotting. This means we don't have to filter out each category ourselves like we did for `sns.distplot`.

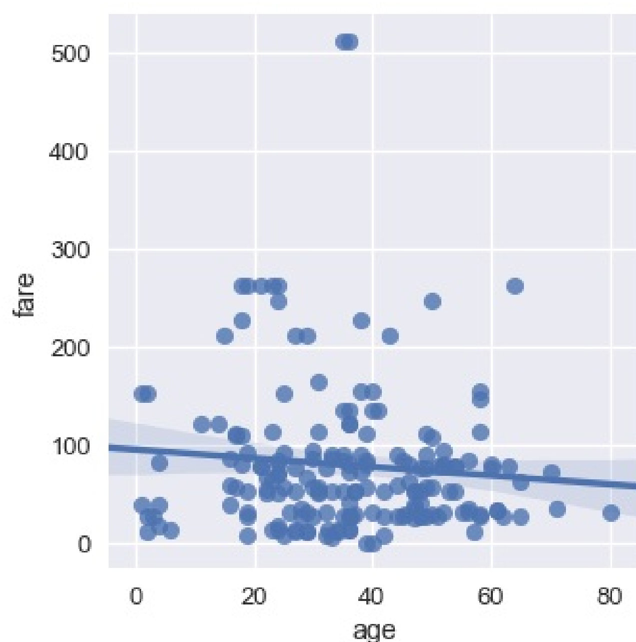
```
# fare (numerical) on the x-axis,
# who (nominal) on the y-axis
sns.boxplot(x='fare', y='who', data=ti);
```



Scatter Plots¶

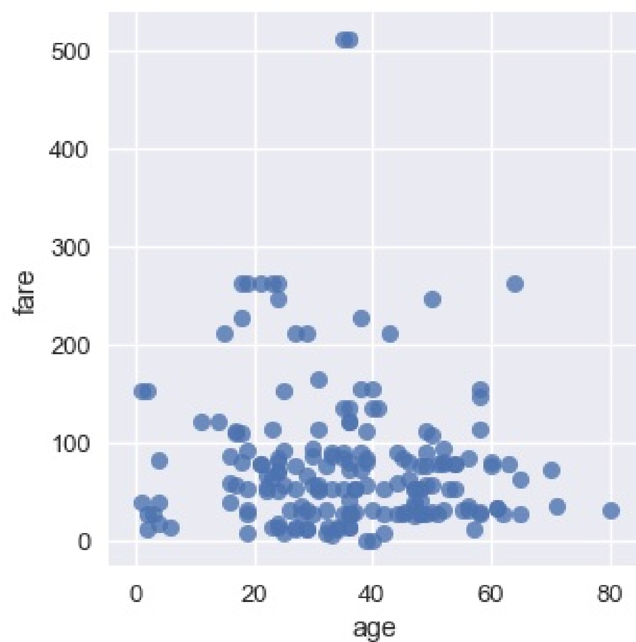
Scatter plots are used to compare two quantitative variables. We can compare the `age` and `fare` columns of our Titanic dataset using a scatter plot.

```
sns.lmplot(x='age', y='fare', data=ti);
```



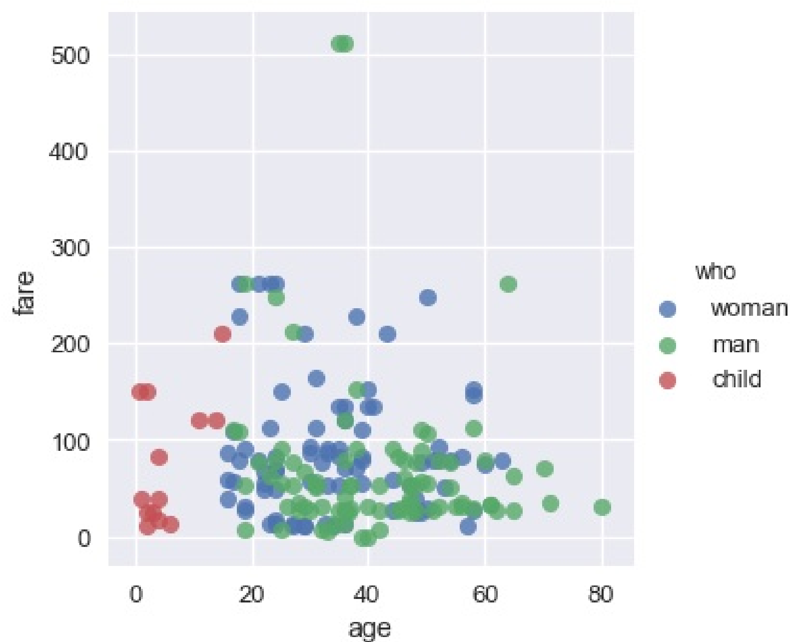
By default seaborn will also fit a regression line to our scatterplot and bootstrap the scatterplot to create a 95% confidence interval around the regression line shown as the light blue shading around the line above. In this case, the regression line doesn't seem to fit the scatter plot very well so we can turn off the regression.

```
sns.lmplot(x='age', y='fare', data=ti, fit_reg=False);
```



We can color the points using a categorical variable. Let's use the `who` column once more:

```
sns.lmplot(x='age', y='fare', hue='who', data=ti,
fit_reg=False);
```



From this plot we can see that all passengers below the age of 18 or so were marked as `child`. There doesn't seem to be a noticable split between male and female passenger fares, although the two most expensive tickets were purchased by males.

Show Widgets Open on DataHub

Table of Contents

Visualizing Qualitative Data¶

For qualitative or categorical data, we most often use bar charts and dot charts. We will show how to create these plots using `seaborn` and the Titanic survivors dataset.

```
# Import seaborn and apply its plotting styles
import seaborn as sns
sns.set()

# Load the dataset
ti = sns.load_dataset('titanic').reset_index(drop=True)

# This table is too large to fit onto a page so we'll output
sliders to
# pan through different sections.
df_interact(ti)
```

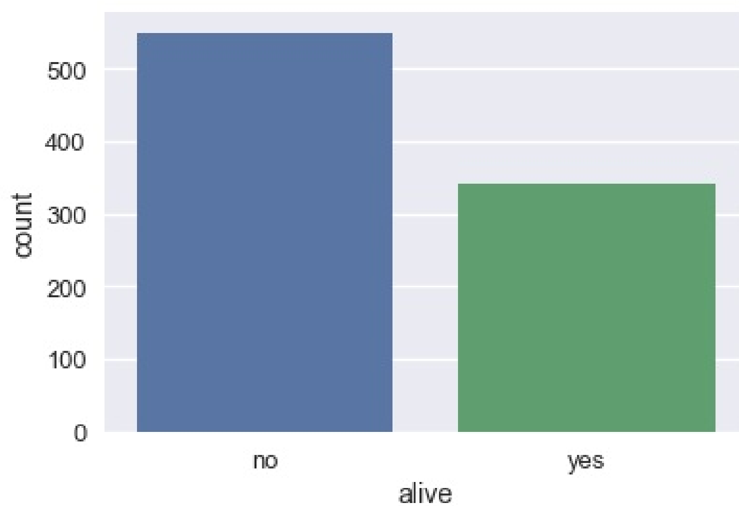
Show Widget

```
(891 rows, 15 columns) total
```

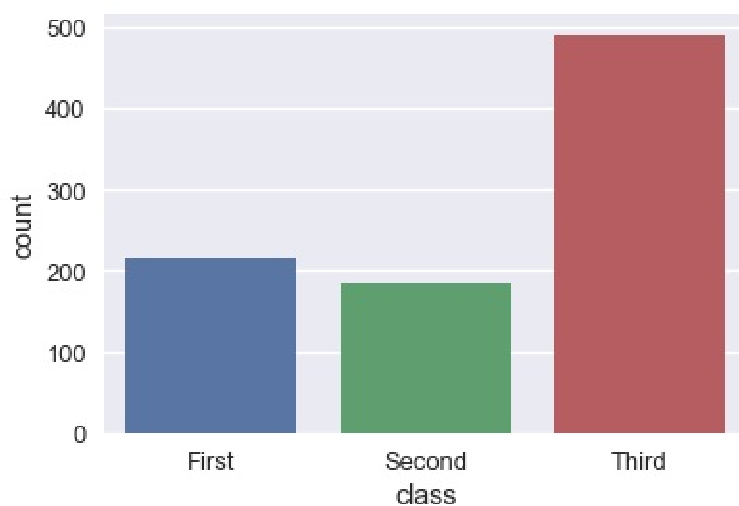
Bar Charts¶

In `seaborn`, there are two types of bar charts. The first type uses the `countplot` method to count up the number of times each category appears in a column.

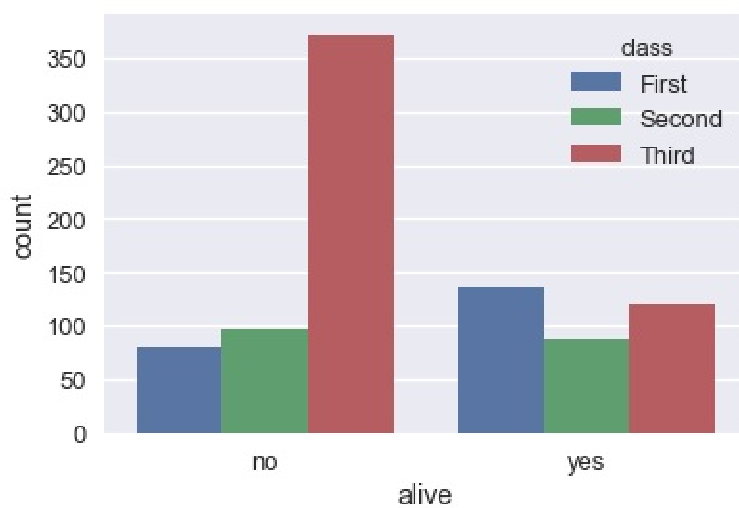
```
# Counts how many passengers survived and didn't survive and
# draws bars with corresponding heights
sns.countplot(x='alive', data=ti);
```




```
sns.countplot(x='class', data=ti);
```

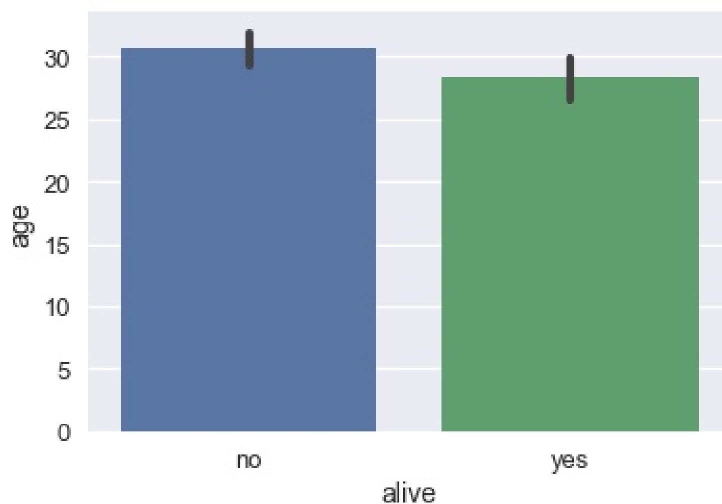



```
# As with box plots, we can break down each category further
using color
sns.countplot(x='alive', hue='class', data=ti);
```



The `barplot` method, on the other hand, groups the DataFrame by a categorical column and plots the height of the bars according to the average of a numerical column within each group.

```
# For each set of alive/not alive passengers, compute and plot
the average age.
sns.barplot(x='alive', y='age', data=ti);
```



The height of each bar can be computed by grouping the original DataFrame and averaging the `age` column:

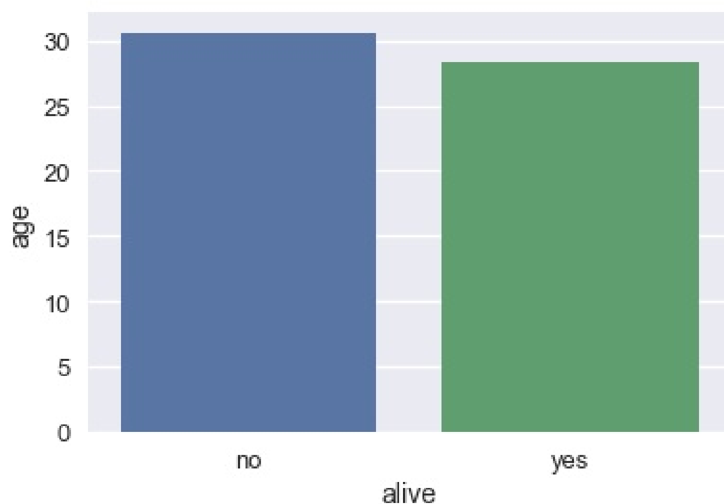
```
ti[['alive', 'age']].groupby('alive').mean()
```

	age
alive	
no	30.626179
yes	28.343690

By default, the `barplot` method will also compute a bootstrap 95% confidence interval for each averaged value, marked as the black lines in the bar chart above. The confidence intervals show that if the dataset contained a random sample of Titanic passengers, the difference between passenger age for those that survived and those that didn't is not statistically significant at the 5% significance level.

These confidence intervals take long to generate when we have larger datasets so it is sometimes useful to turn them off:

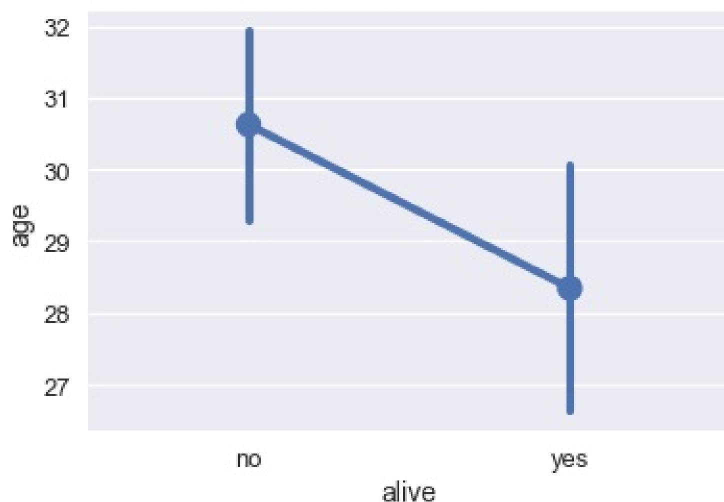
```
sns.barplot(x='alive', y='age', data=ti, ci=False);
```



Dot Charts¶

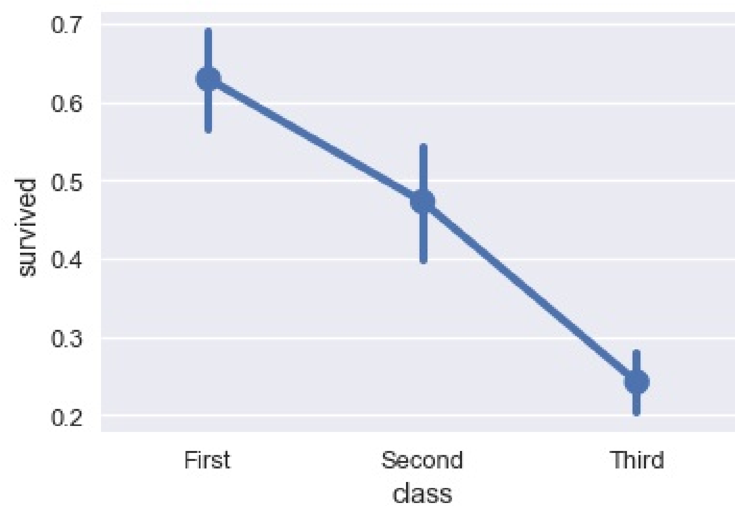
Dot charts are similar to bar charts. Instead of plotting bars, dot charts mark a single point at the end of where a bar would go. We use the `pointplot` method to make dot charts in `seaborn`. Like the `barplot` method, the `pointplot` method also automatically groups the DataFrame and computes the average of a separate numerical variable, marking 95% confidence intervals as vertical lines centered on each point.

```
# For each set of alive/not alive passengers, compute and plot
the average age.
sns.pointplot(x='alive', y='age', data=ti);
```

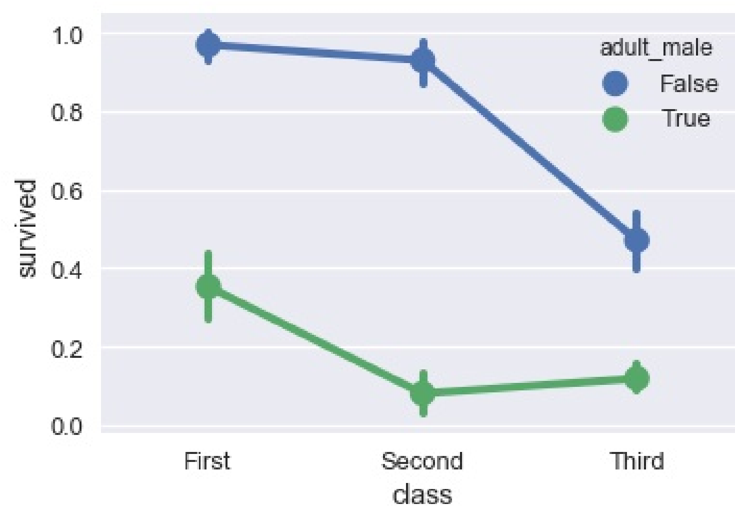


Dot charts are most useful when comparing changes across categories:

```
# Shows the proportion of survivors for each passenger class
sns.pointplot(x='class', y='survived', data=ti);
```




```
# Shows the proportion of survivors for each passenger class,
# split by whether the passenger was an adult male
sns.pointplot(x='class', y='survived', hue='adult_male',
data=ti);
```



Show Widgets Open on DataHub

Table of Contents

Customizing Plots using `matplotlib` ¶

Although `seaborn` allows us to quickly create many types of plots, it does not give us fine-grained control over the chart. For example, we cannot use `seaborn` to modify a plot's title, change x or y-axis labels, or add annotations to a plot. Instead, we must use the `matplotlib` library that `seaborn` is based off of.

`matplotlib` provides basic building blocks for creating plots in Python. Although it gives great control, it is also more verbose—recreating the `seaborn` plots from the previous sections in `matplotlib` would take many lines of code. In fact, we can think of `seaborn` as a set of useful shortcuts to create `matplotlib` plots. Although we prefer to prototype plots in `seaborn`, in order to customize plots for publication we will need to learn basic pieces of `matplotlib`.

Before we look at our first simple example, we must activate matplotlib support in the notebook:

```
# This line allows matplotlib plots to appear as images in the notebook
# instead of in a separate window.
%matplotlib inline

# plt is a commonly used shortcut for matplotlib
import matplotlib.pyplot as plt
```

Customizing Figures and Axes¶

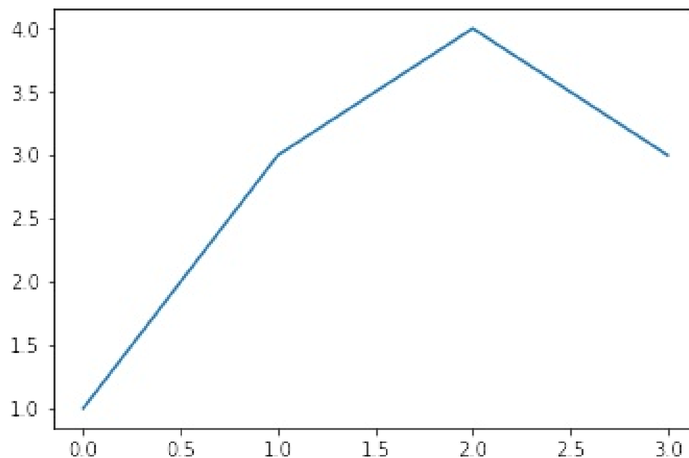
In order to create a plot in `matplotlib`, we create a *figure*, then add an *axes* to the figure. In `matplotlib`, an axes is a single chart, and figures can contain multiple axes in a tablular layout. An axes contains *marks*, the lines or patches drawn on the plot.

```
# Create a figure
f = plt.figure()

# Add an axes to the figure. The second and third arguments
create a table
# with 1 row and 1 column. The first argument places the axes in
the first
# cell of the table.
ax = f.add_subplot(1, 1, 1)

# Create a line plot on the axes
ax.plot([0, 1, 2, 3], [1, 3, 4, 3])

# Show the plot. This will automatically get called in a Jupyter
notebook
# so we'll omit it in future cells
plt.show()
```



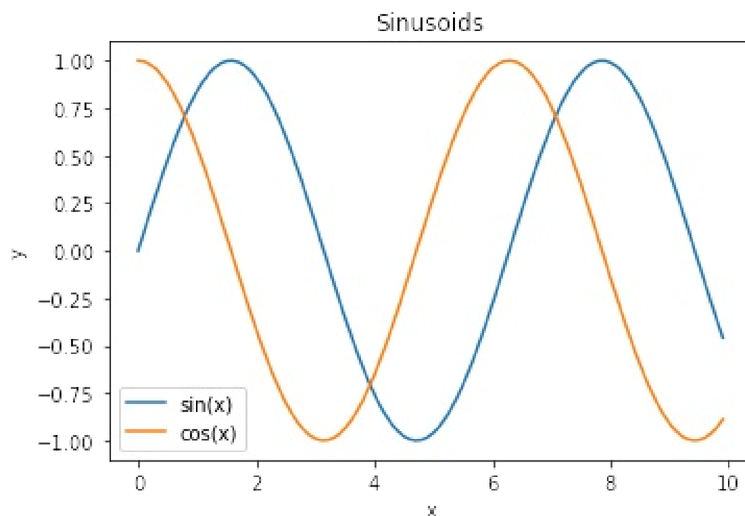
To customize the plot, we can use other methods on the axes object:

```
f = plt.figure()
ax = f.add_subplot(1, 1, 1)

x = np.arange(0, 10, 0.1)

# Setting the label kwarg lets us generate a legend
ax.plot(x, np.sin(x), label='sin(x)')
ax.plot(x, np.cos(x), label='cos(x)')
ax.legend()

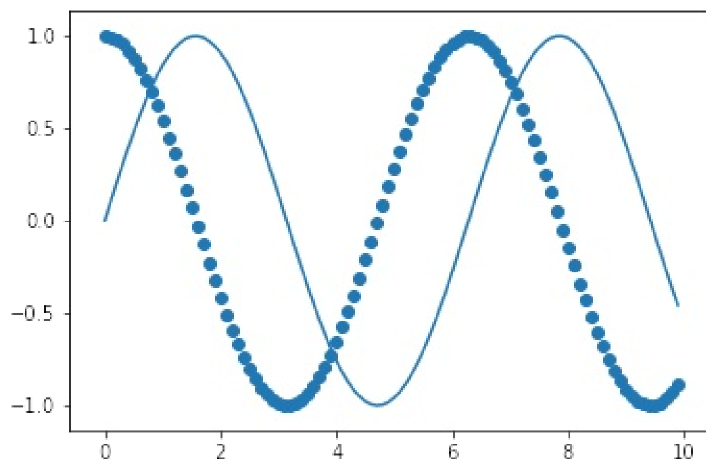
ax.set_title('Sinusoids')
ax.set_xlabel('x')
ax.set_ylabel('y');
```



As a shortcut, `matplotlib` has plotting methods on the `plt` module itself that will automatically initialize a figure and axes.

```
# Shorthand to create figure and axes and call ax.plot
plt.plot(x, np.sin(x))

# When plt methods are called multiple times in the same cell,
the
# existing figure and axes are reused.
plt.scatter(x, np.cos(x));
```



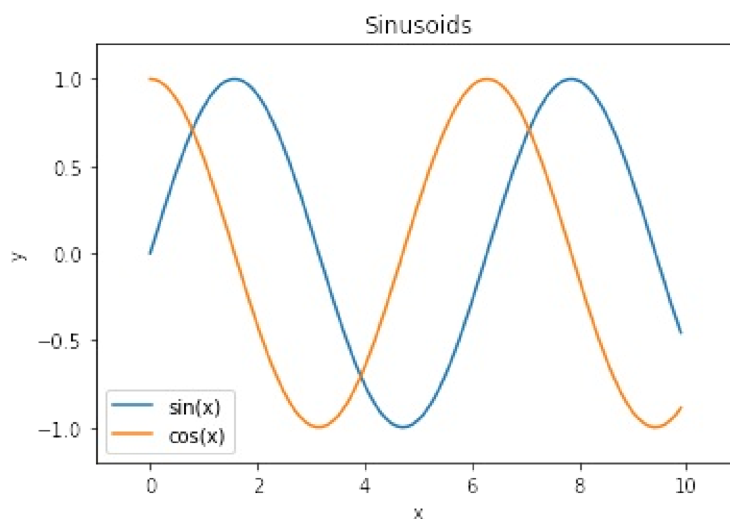
The `plt` module has analogous methods to an axes, so we can recreate one of the plots above using `plt` shorthands.

```
x = np.arange(0, 10, 0.1)

plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend()

# Shorthand for ax.set_title
plt.title('Sinusoids')
plt.xlabel('x')
plt.ylabel('y')

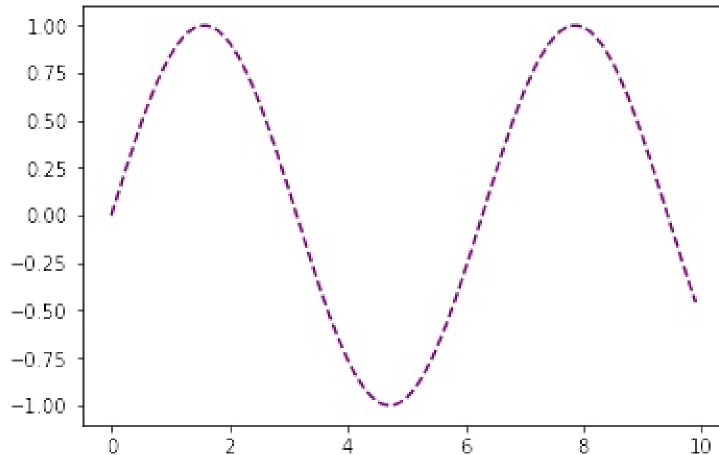
# Set the x and y-axis limits
plt.xlim(-1, 11)
plt.ylim(-1.2, 1.2);
```



Customizing Marks¶

To change properties of the plot marks themselves (e.g. the lines in the plot above), we can pass additional arguments into `plt.plot` .

```
plt.plot(x, np.sin(x), linestyle='--', color='purple');
```



Checking the `matplotlib` documentation is the easiest way to figure out which arguments are available for each method. Another way is to store the returned line object:

```
In [1]: line, = plot([1,2,3])
```

These line objects have a lot of properties you can control, here's the full list using tab-completion in IPython:

```
In [2]: line.set
line.set                     line.set_drawstyle           line.set_mec
line.set_aa                  line.set_figure              line.set_mew
line.set_agg_filter          line.set_fillstyle           line.set_mfc
line.set_alpha               line.set_gid                 line.set_mfcalt
line.set_animated            line.set_label               line.set_ms
line.set_antialiased         line.set_linestyle           line.set_picker
line.set_axes                line.set_linewidth           line.set_pickradius
line.set_c                   line.set_lod                 line.set_rasterized
line.set_clip_box            line.set_ls                  line.set_snap
line.set_clip_on             line.set_lw                  line.set_solid_capstyle
line.set_clip_path           line.set_marker              line.set_solid_joinstyle
line.set_color               line.set_markeredgecolor     line.set_transform
line.set_contains            line.set_markeredgewidth     line.set_url
line.set_dash_capstyle       line.set_markerfacecolor     line.set_visible
line.set_dashes              line.set_markerfacecoloralt  line.set_xdata
line.set_dash_joinstyle      line.set_markersize          line.set_ydata
line.set_data                line.set_markevery           line.set_zorder
```

But the `setp` call (short for set property) can be very useful, especially while working interactively because it contains introspection support, so you can learn about the valid calls as you work:

```
In [7]: line, = plot([1,2,3])

In [8]: setp(line, 'linestyle')
  linestyle: [ ``'-'`` | ``'--'`` | ``'-.'`` | ``':'`` | ``'None'`` | ``' '`` | ``''``
 ]         and any drawstyle in combination with a linestyle, e.g. ``'steps--'``.


In [9]: setp(line)
  agg_filter: unknown
  alpha: float (0.0 transparent through 1.0 opaque)
  animated: [True | False]
  antialiased or aa: [True | False]
  ...
  ... much more output omitted
  ...
```

In the first form, it shows you the valid values for the 'linestyle' property, and in the second it shows you all the acceptable properties you can set on the line object. This makes it easy to discover how to customize your figures to get the visual results you need.

Aribitrary text and LaTeX support¶

In matplotlib, text can be added either relative to an individual axis object or to the whole figure.

These commands add text to the Axes:

- `set_title()` - add a title
- `set_xlabel()` - add an axis label to the x-axis
- `set_ylabel()` - add an axis label to the y-axis
- `text()` - add text at an arbitrary location
- `annotate()` - add an annotation, with optional arrow

And these act on the whole figure:

- `figtext()` - add text at an arbitrary location
- `suptitle()` - add a title

And any text field can contain LaTeX expressions for mathematics, as long as they are enclosed in `$` signs.

This example illustrates all of them:

```
fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14,
fontweight='bold')

ax = fig.add_subplot(1, 1, 1)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords',
style='italic',
        bbox={'facecolor':'red', 'alpha':0.5, 'pad':10})

ax.text(2, 6, 'an equation: $E=mc^2$', fontsize=15)

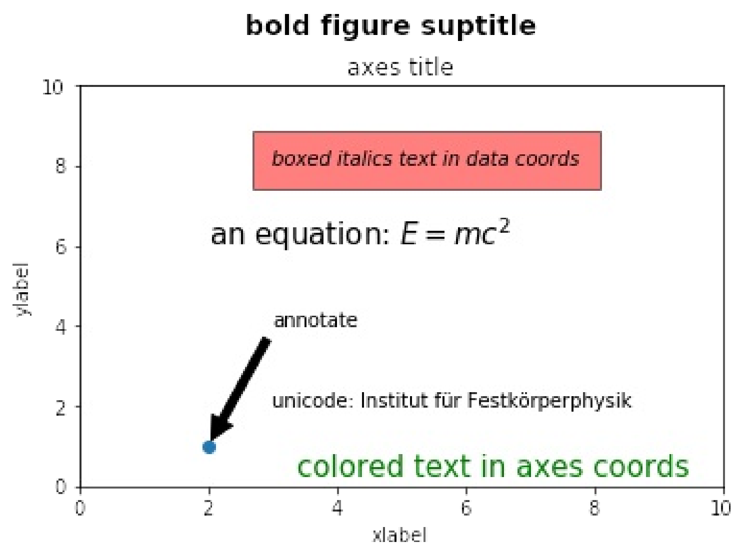
ax.text(3, 2, 'unicode: Institut für Festkörperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)



ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10]);
```



Customizing a `seaborn` plot using `matplotlib` ¶

Now that we've seen how to use `matplotlib` to customize a plot, we can use the same methods to customize `seaborn` plots since `seaborn` creates plots using `matplotlib` behind-the-scenes.

```
# Load seaborn
import seaborn as sns
sns.set()
sns.set_context('talk')

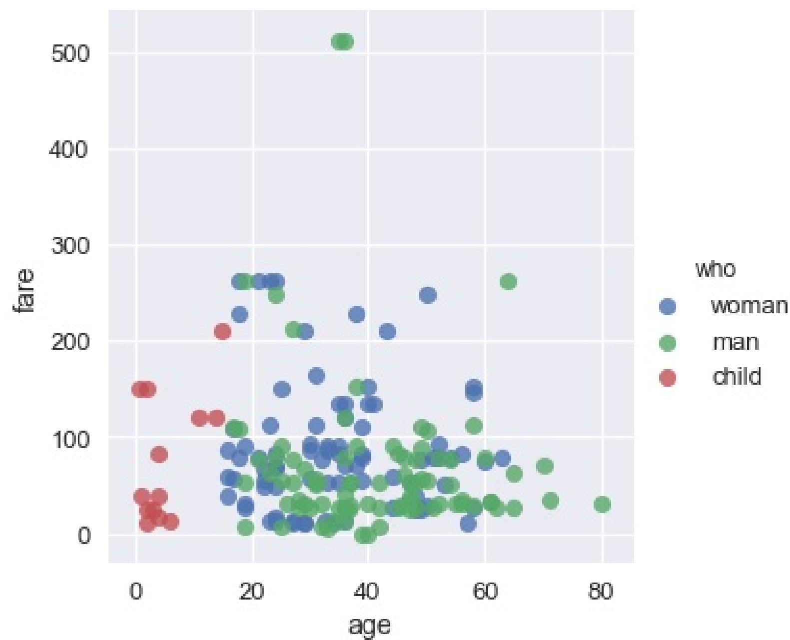
# Load dataset
ti = sns.load_dataset('titanic').dropna().reset_index(drop=True)
ti.head()
```

	survived	pclass	sex	age	...	deck	embark_town	alive
0	1	1	female	38.0	...	C	Cherbourg	yes
1	1	1	female	35.0	...	C	Southampton	yes
2	0	1	male	54.0	...	E	Southampton	no
3	1	3	female	4.0	...	G	Southampton	yes
4	1	1	female	58.0	...	C	Southampton	yes

5 rows × 15 columns

We'll start with this plot:

```
sns.lmplot(x='age', y='fare', hue='who', data=ti,
fit_reg=False);
```

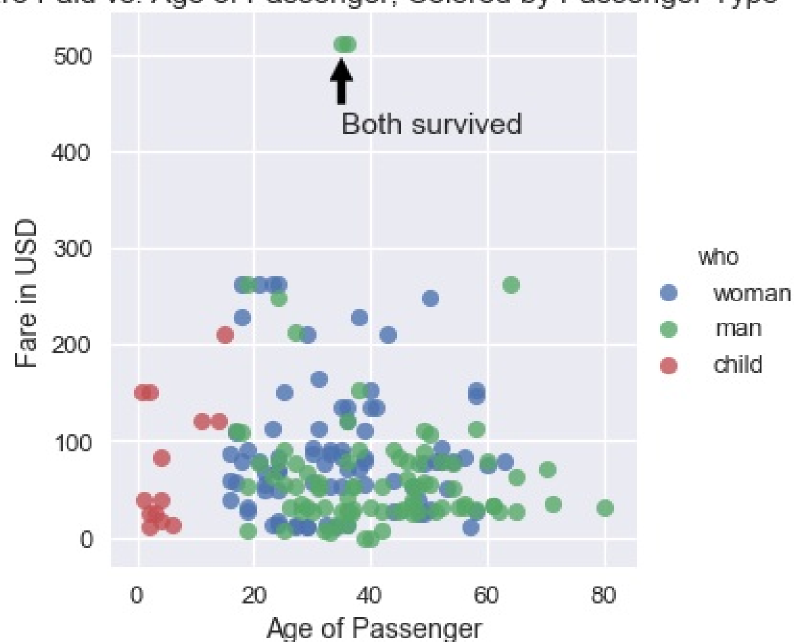


We can see that the plot needs a title and better labels for the x and y-axes. In addition, the two people with the most expensive fares survived, so we can annotate them on our plot.

```
sns.lmplot(x='age', y='fare', hue='who', data=ti, fit_reg=False)

plt.title('Fare Paid vs. Age of Passenger, Colored by Passenger
Type')
plt.xlabel('Age of Passenger')
plt.ylabel('Fare in USD')

plt.annotate('Both survived', xy=(35, 500), xytext=(35, 420),
             arrowprops=dict(facecolor='black', shrink=0.05));
```



In practice, we use `seaborn` to quickly explore the data and then turn to `matplotlib` for fine-tuning once we decide on the plots to use in a paper or presentation.

Show Widgets Open on DataHub

Table of Contents

Visualization Principles¶

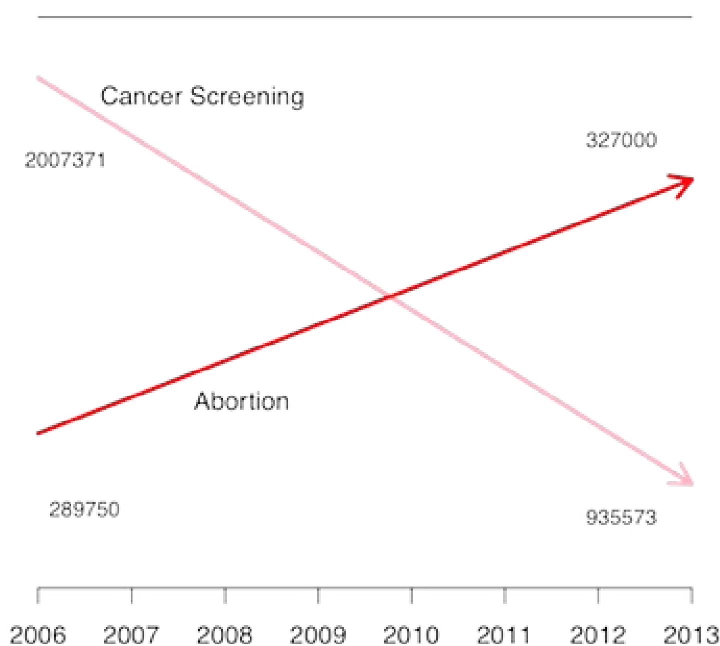
Now that we have the tools to create and alter plots, we turn to key principles for data visualization. Much like other parts of data science, it is difficult to precisely assign a number that measures how effective a specific visualization is. Still, there are general principles that make visualizations much more effective at showing trends in the data. We discuss six categories of principles: scale, conditioning, perception, transformation, context, and smoothing.

Principles of Scale¶

Principles of scale relate to the choice of x and y-axis used to plot the data.

In a 2015 US Congressional hearing, representative Chaffetz discussed an investigation of Planned Parenthood programs. He presented the following plot that originally appeared in a report by Americans United for Life. It compares the number of abortion and cancer screening procedures, both of which are offered by Planned Parenthood. (The full report is available at https://oversight.house.gov/interactivepage/plannedparenthood.)

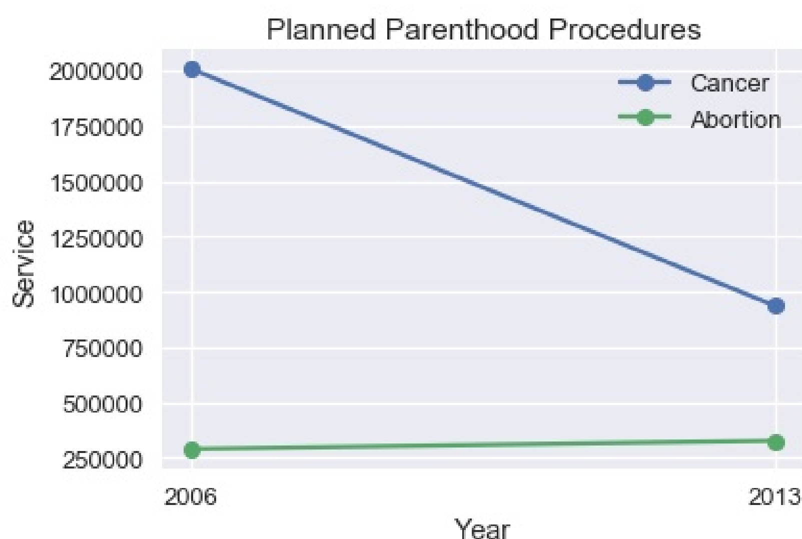
What is suspicious about this plot? How many data points are plotted?



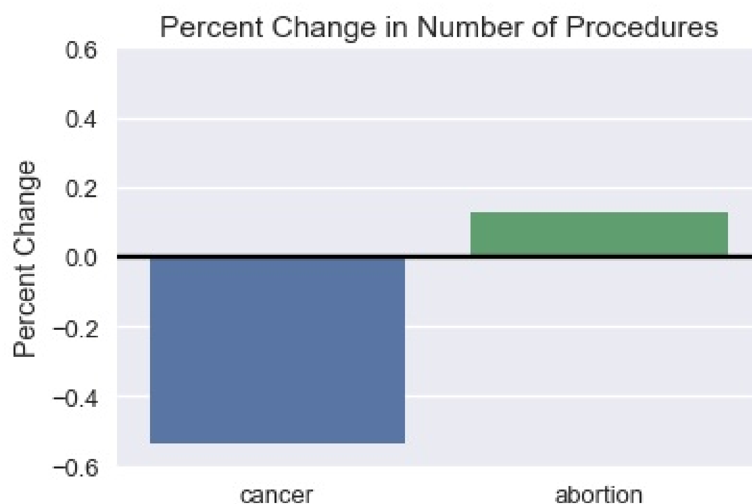
This plot violates principles of scale; it doesn't make good choices for its x and y-axis.

When we select the x and y-axis for our plot, we should keep a consistent scale across the entire axis. However, the plot above has different scales for the Abortion and Cancer Screening lines—the start of the Abortion line and end of the Cancer Screening line lie close to each other on the y-axis but represent vastly different numbers. In addition, only points from 2006 and 2013 are plotted but the x-axis contains unnecessary tick marks for every year in between.

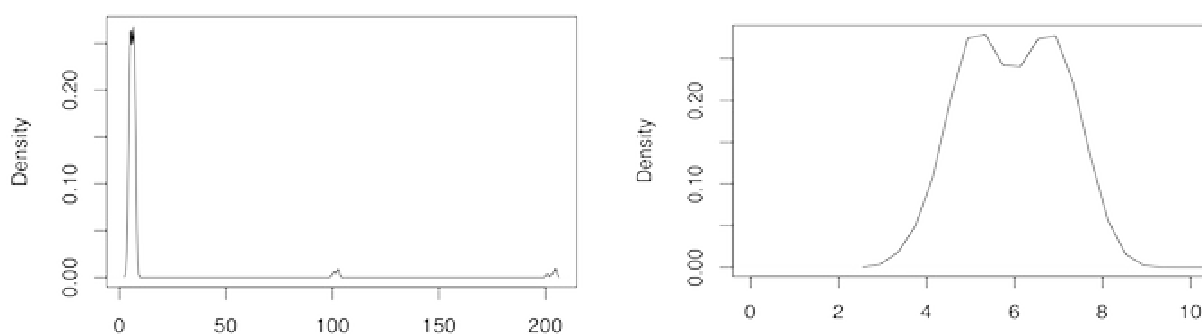
To improve this plot, we should re-plot the points on the same y-axis scale:



We can see that the change in number of Abortions is very small compared to the large drop in the number of Cancer Screenings. Instead of the number of procedures, we might instead be interested in the percent change in number.



When selecting the x and y-axis limits we prefer to focus on the region with the bulk of the data, especially when working with long-tailed data. Consider the following plot and its zoomed in version to its right:



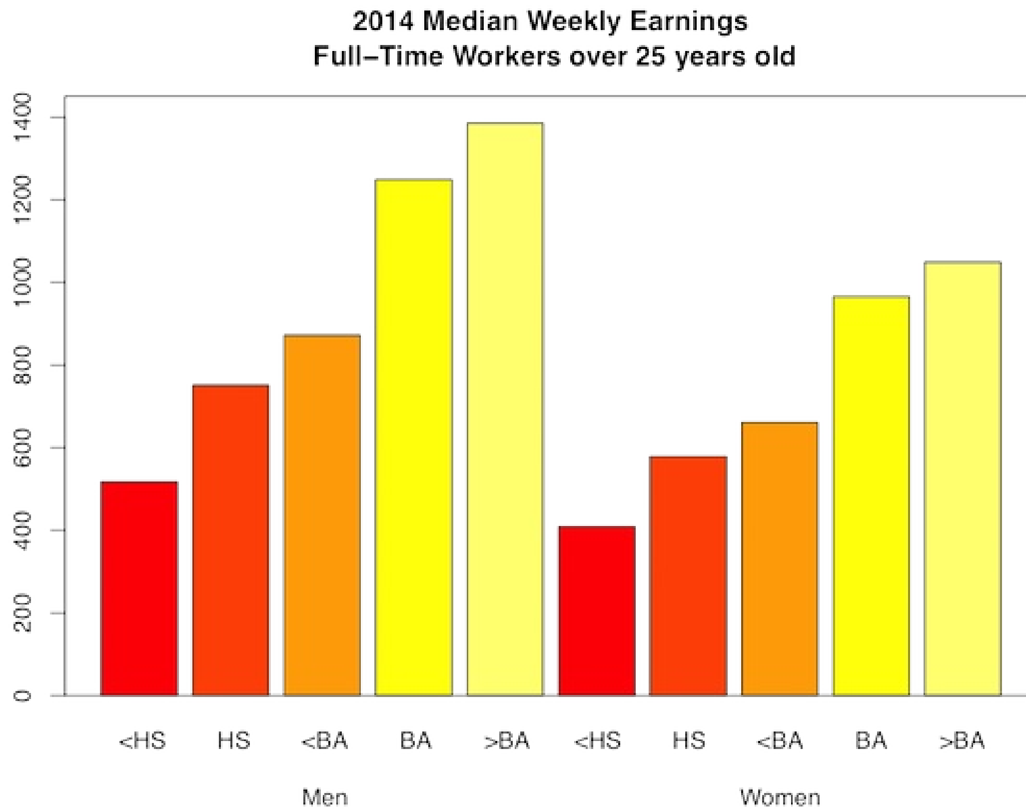
The plot on the right is much more helpful for making sense of the dataset. If needed, we can make multiple plots of different regions of the data to show the entire range of data. Later in this section, we discuss data transformations which also help visualize long-tailed data.

Principles of Conditioning¶

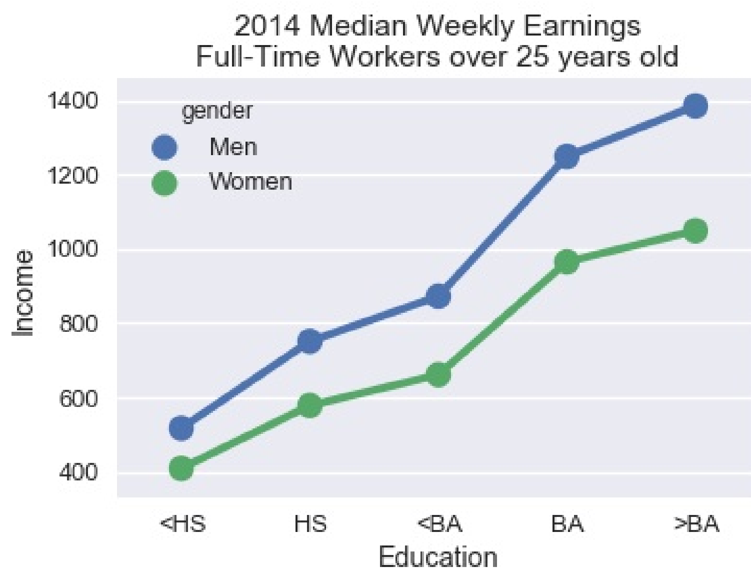
Principles of conditioning give us techniques to show distributions and relationships between subgroups of our data.

The US Bureau of Labor Statistics oversees scientific surveys related to the economic health of the US. Their website contains a tool to generate reports using this data that was used to generate this chart comparing median weekly earnings split by sex.

Which comparisons are easiest to make using this plot? Are these the comparisons that are most interesting or important?



This plot lets us see at a glance that weekly earnings tend to increase with more education. However, it is difficult to tell exactly how much each level of education increases earnings and it is even more difficult to compare male and female weekly earnings at the same education level. We can uncover both these trends by using a dot chart instead of a bar chart.

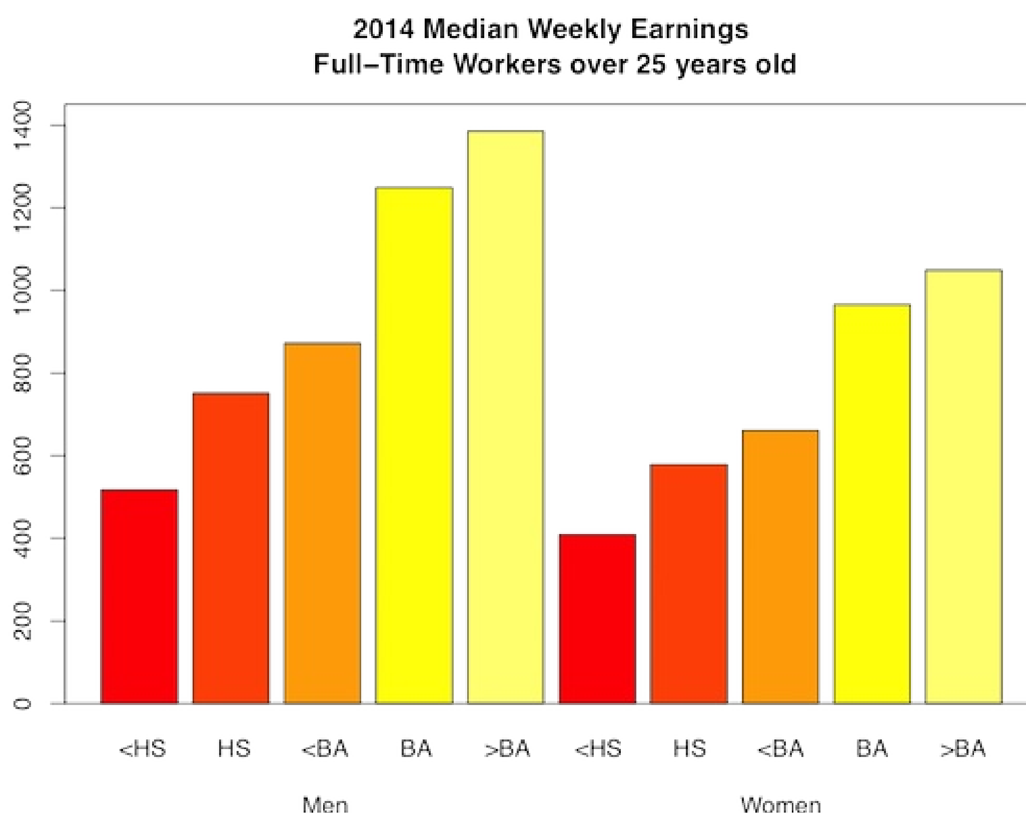


The lines connecting the points more clearly shows the relatively large effect of having a BA degree on weekly earnings. Placing the points for males and females directly above each other makes it much easier to see that the wage gap between males and females tends to increase with higher education levels.

To aid comparison of two subgroups within your data, align markers along the x or y-axis and use different colors or markers for different subgroups. Lines tend to show trends in data more clearly than bars and are a useful choice for both ordinal and numerical data.

Principles of Perception¶

Human perception has specific properties that are important to consider in visualization design. The first important property of human perception is that we perceive some colors more strongly than others, especially green colors. In addition, we perceive lighter shaded areas as larger than darker shaded ones. For example, in the weekly earnings plot that we just discussed, the lighter bars seem to draw more attention than the darker colored ones:

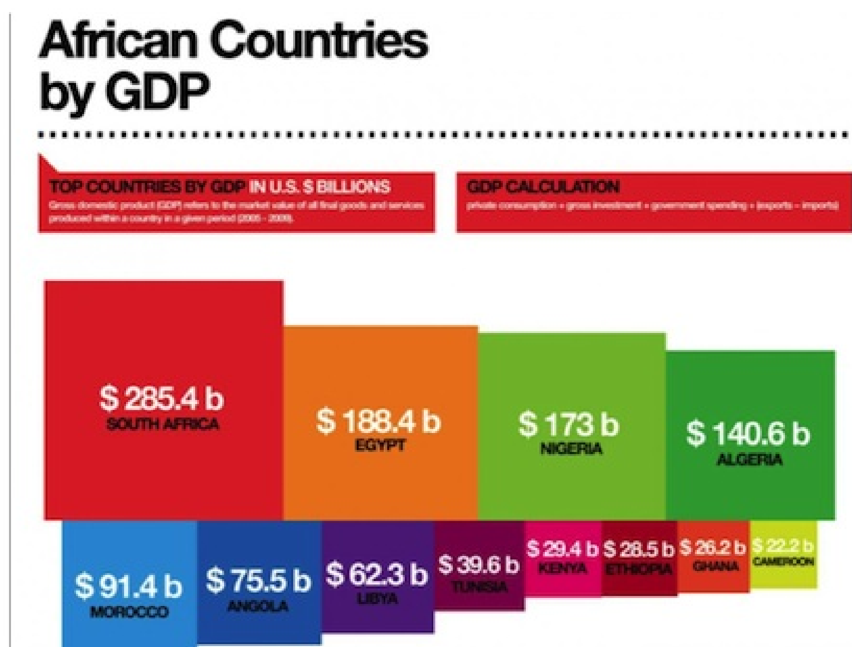


Practically speaking, you should ensure that your charts' color palettes are *perceptually uniform*. This means that, for example, the perceived intensity of the color won't change in between bars in a bar chart. For quantitative data, you have two choices: if your data progress from low to high and you want to emphasize large values, use a *sequential* color

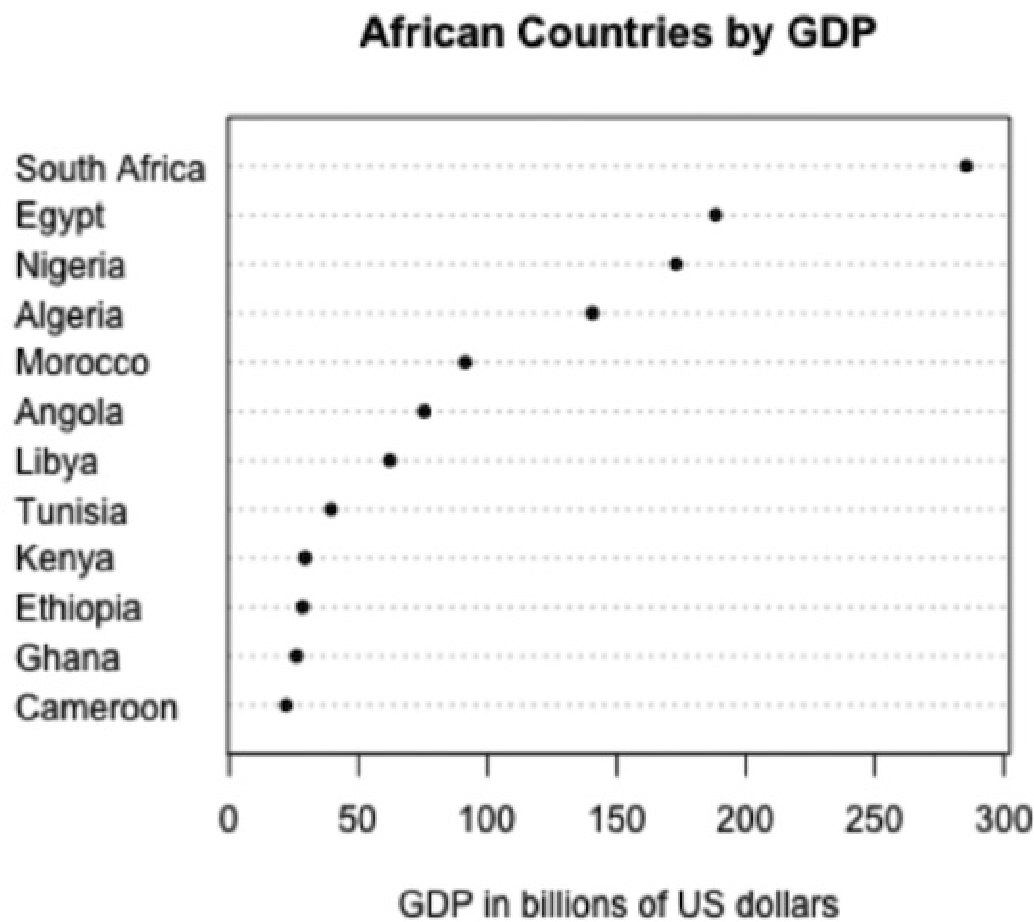
scheme which assigns lighter colors to large values. If both low and high values should be emphasized, use a *diverging* color scheme which assigns ligher colors to values closer to the center.

`seaborn` comes with many useful color palettes built-in. You can browse its documentation to learn how to switch between color palettes:
http://seaborn.pydata.org/tutorial/color_palettes.html

A second important property of human perception is that we are generally more accurate when we compare lengths and less accurate when we compare areas. Consider the following chart of the GDP of African countries.

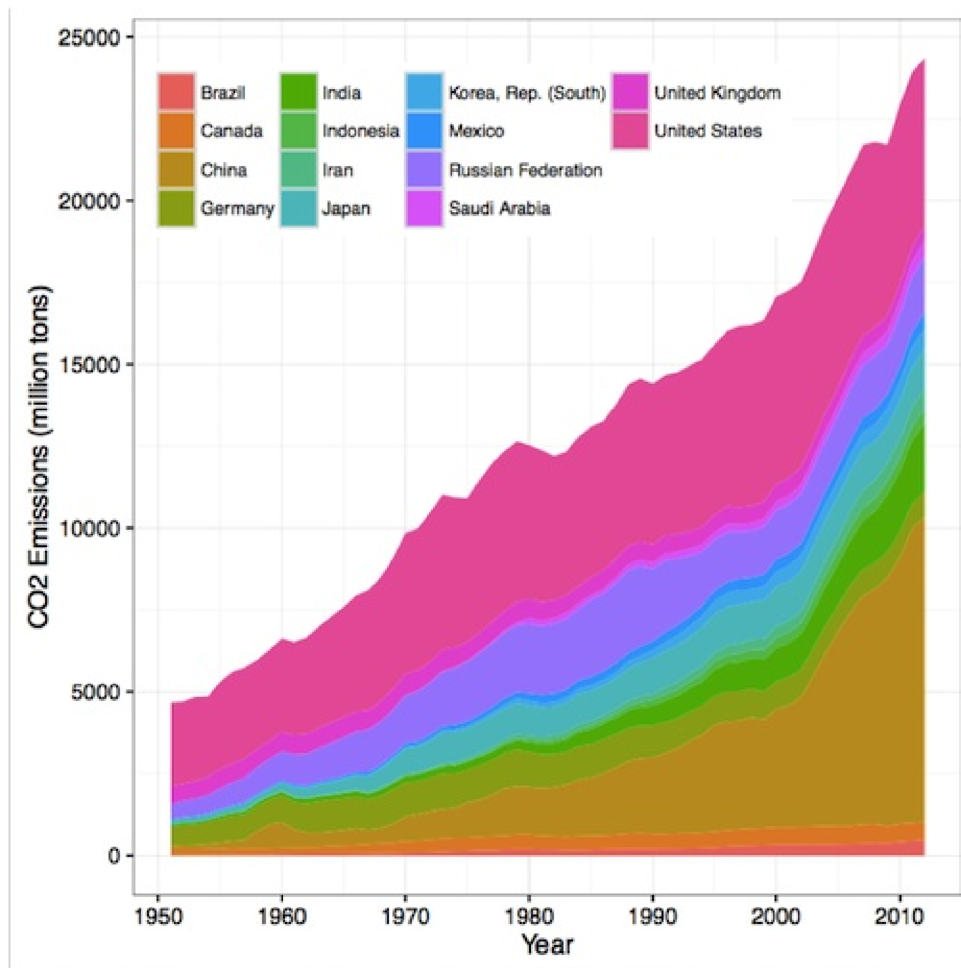


By numerical value, South Africa has twice the GDP of Algeria but it's not easy to tell from the plot above. Instead, we can plot the GDPs on a dot plot:



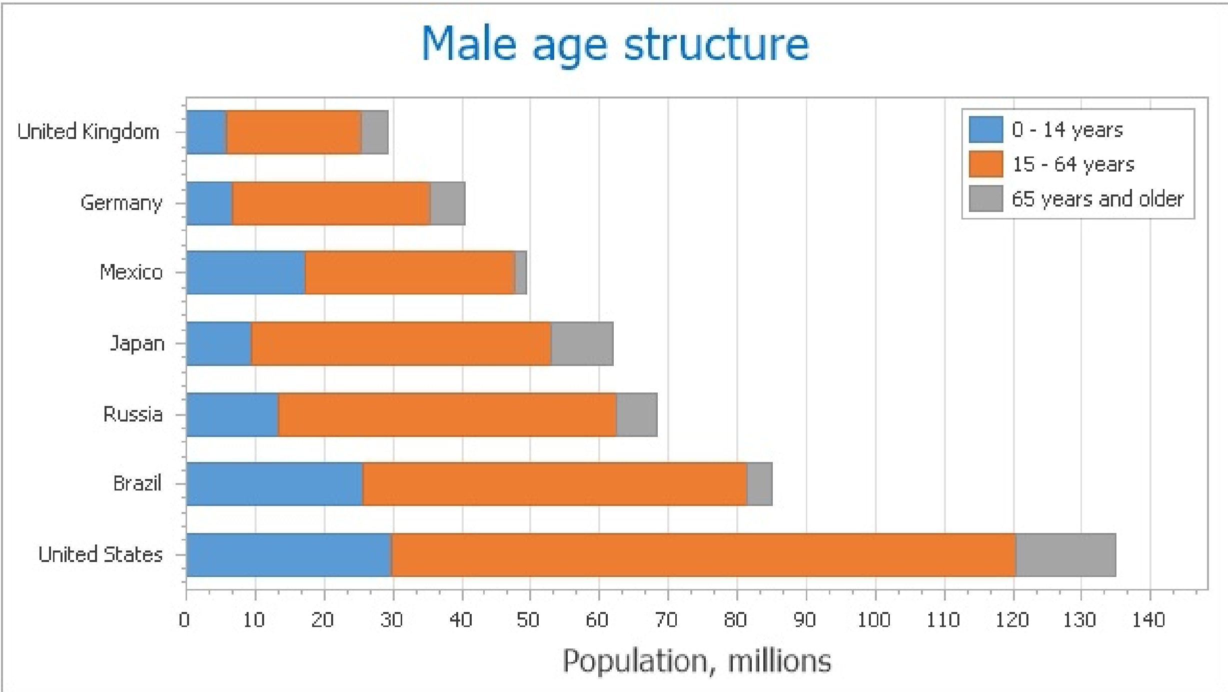
This is much more clear because it allows us to compare lengths instead of areas. Pie charts and three-dimensional charts are difficult to interpret for the same reason; we tend to avoid these charts in practice.

Our third and final property of perception is that the human eye has difficulty with changing baselines. Consider the following stacked area chart that plots carbon dioxide emissions over time split by country.

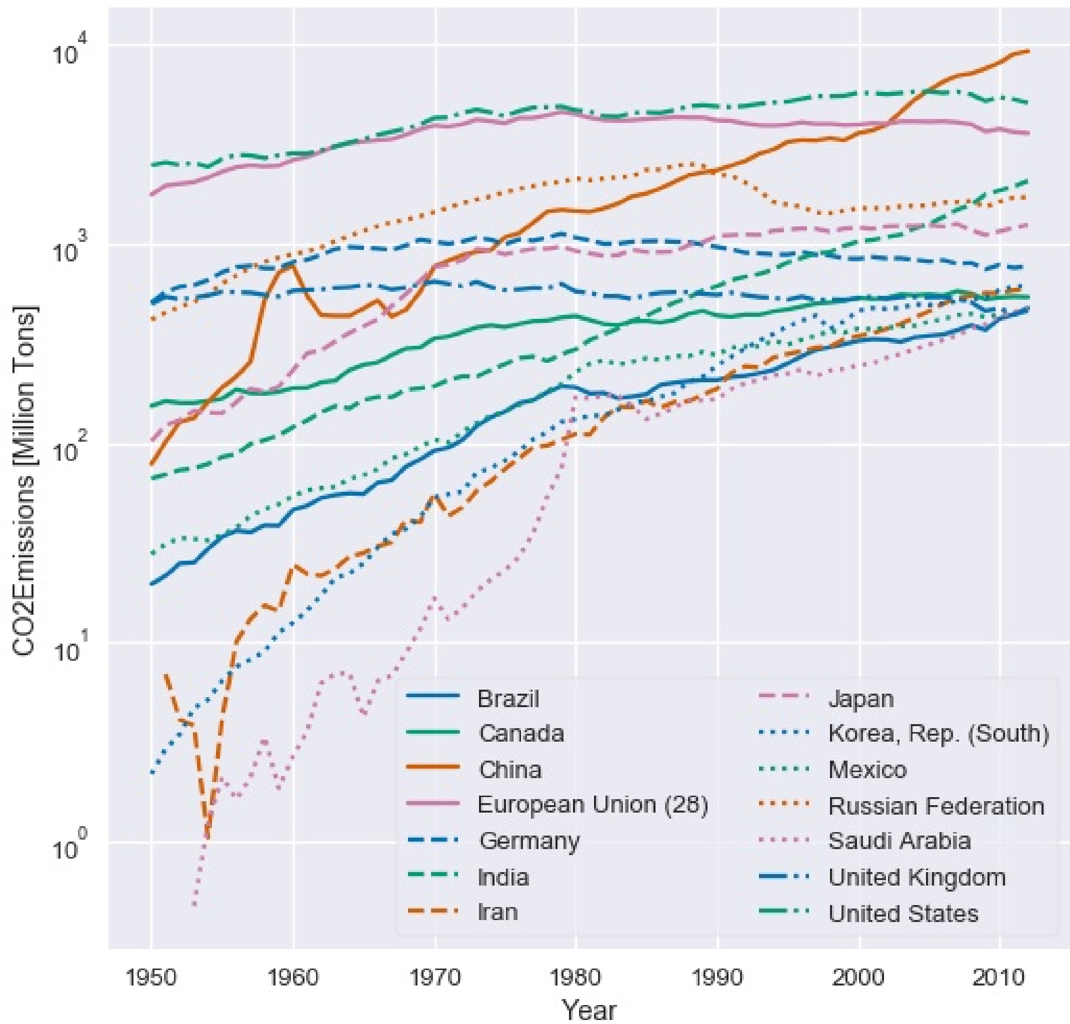


It is difficult to see whether the UK's emissions have increased or decreased over time because the of *jiggling baseline* problem: the baseline (bottom line) of the area jiggles up and down. It is also difficult to compare whether the UK's emissions are greater than China's emissions when the two heights are similar (in year 2000, for example).

Similar issues of jiggling baselines appear in stacked bar charts. In the plot below, it is difficult to compare the number of 15-64 year olds between Germany and Mexico.



We can often improve a stacked area or bar chart by switching to a line chart. Here's the data of emissions over time plotted as lines instead of areas:



This plot does not jiggle the baseline so it is much easier to compare emissions between countries. We can also more clearly see which countries increased emissions the most.

Show Widgets Open on DataHub

Table of Contents

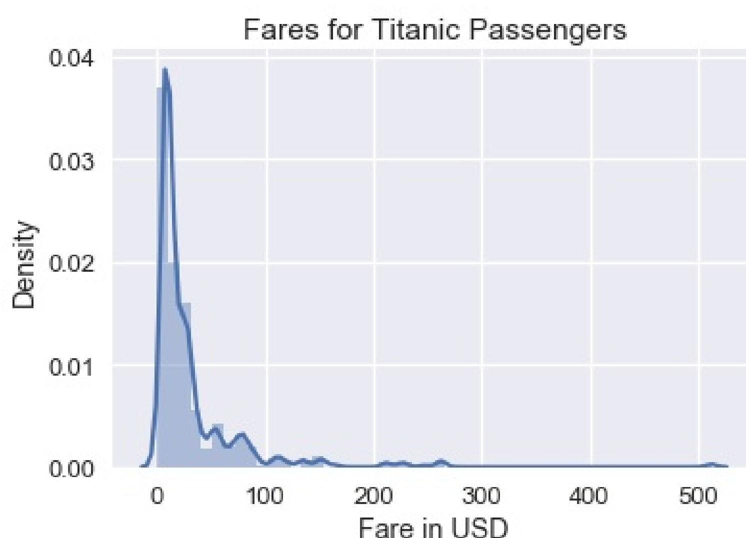
Visualization Principles Continued¶

In this section, we discuss principles of visualization for transformation, context, and smoothing.

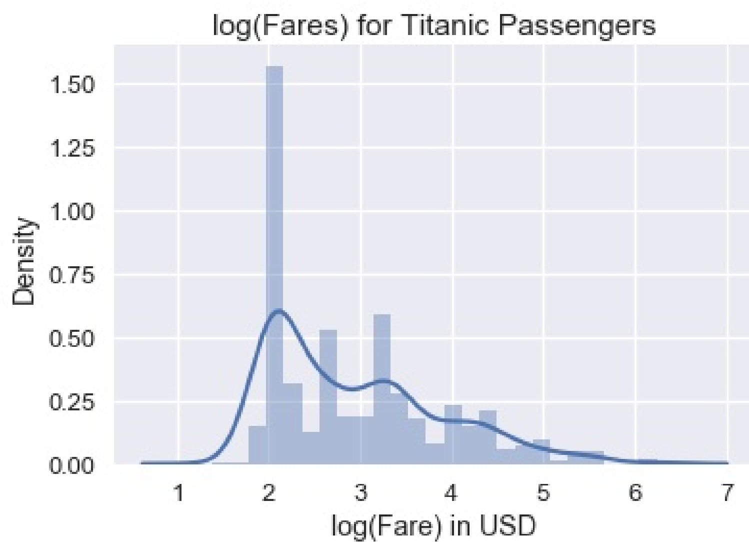
Principles of Transformation¶

The principles of data transformation give us useful ways to alter data for visualization in order to more effectively reveal trends. We most commonly apply data transformations to reveal patterns in skewed data and non-linear relationships between variables.

The plot below shows the distribution of ticket fares for each passenger aboard the Titanic. As you can see, the distribution is skewed right.



Although this histogram shows the all the fares, it is difficult to see detailed patterns in the data since the fares are clumped on the left side of the histogram. To remedy this, we can take the natural log of the fares before plotting them:



We can see from the plot of the log data that the distribution of fares has a mode at roughly $ e^2 = \$7.40 $ and a smaller mode at roughly $ e^{3.4} = \$30.00 $. Why does plotting the natural log of the data help with skew? The logarithms of large numbers tend be close to the logarithms of small numbers:

value	log(value)
1	0.00
10	2.30
50	3.91
100	4.60
500	6.21
1000	6.90

This means that taking the logarithm of right-tailed data will bring large values close to small values. This helps see patterns where the majority of the data lie.

In fact, the logarithm is considered the Swiss army knife of data tranformation—it also helps us see the nature of non-linear relationships between variables in the data. In 1619, Kepler recorded down the following set of data to discover his Third Law of Planetary Motion:

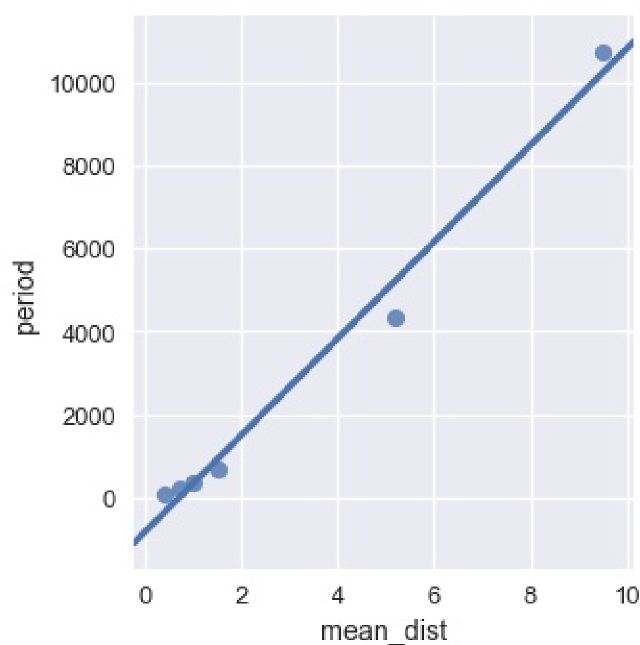
```
planets = pd.read_csv("data/planets.data",
delim_whitespace=True,
                      comment="#", usecols=[0, 1, 2])
planets
```

	planet	mean_dist	period
0	Mercury	0.389	87.77
1	Venus	0.724	224.70
2	Earth	1.000	365.25
3	Mars	1.524	686.95
4	Jupiter	5.200	4332.62
5	Saturn	9.510	10759.20

If we plot the mean distance to the sun against the period of the orbit, we can see a relationship that doesn't quite look linear:

```
sns.lmplot(x='mean_dist', y='period', data=planets, ci=False)
```

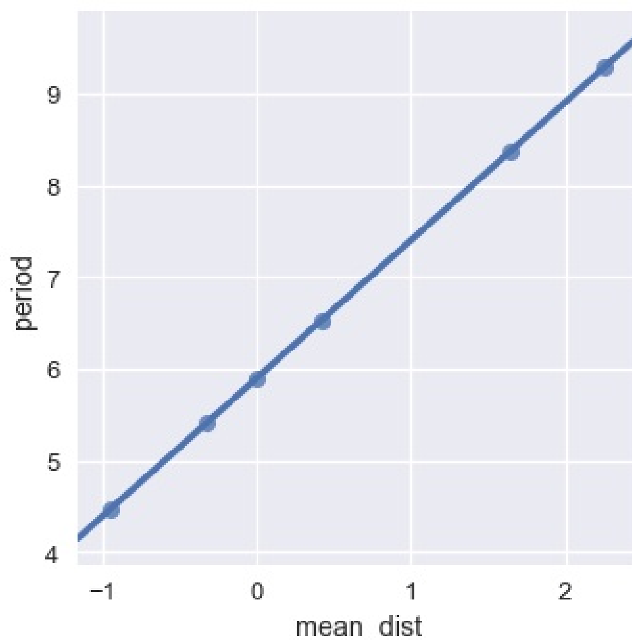
```
<seaborn.axisgrid.FacetGrid at 0x1a1f54aba8>
```



However, if we take the natural log of both mean distance and period, we obtain the following plot:

```
sns.lmplot(x='mean_dist', y='period',
           data=np.log(planets.iloc[:, [1, 2]]),
           ci=False);
```

```
<seaborn.axisgrid.FacetGrid at 0x1a1f693da0>
```



We see a near-perfect linear relationship between the logged values of mean distance and period. What does this mean? Since we believe there's a linear relationship between the logged values, we can derive:

$$ \begin{aligned} \log(period) &= m \log(dist) + b \\ period &= e^{m \log(dist) + b} & \text{Taking the exponent of both sides} \\ period &= e^b dist^m \\ period &= C \cdot dist^m \end{aligned} $$

We replaced $ e^b $ with $ C $ in the last step to represent $ e^b $ as a constant. The algebraic manipulation above shows that when two variables have a polynomial relationship, the log of the two variables has a linear relationship. In fact, we can find the degree of the polynomial by examining the slope of the line. In this case, the slope is 1.5 which gives us Kepler's third law: $ period \propto dist^{1.5} $.

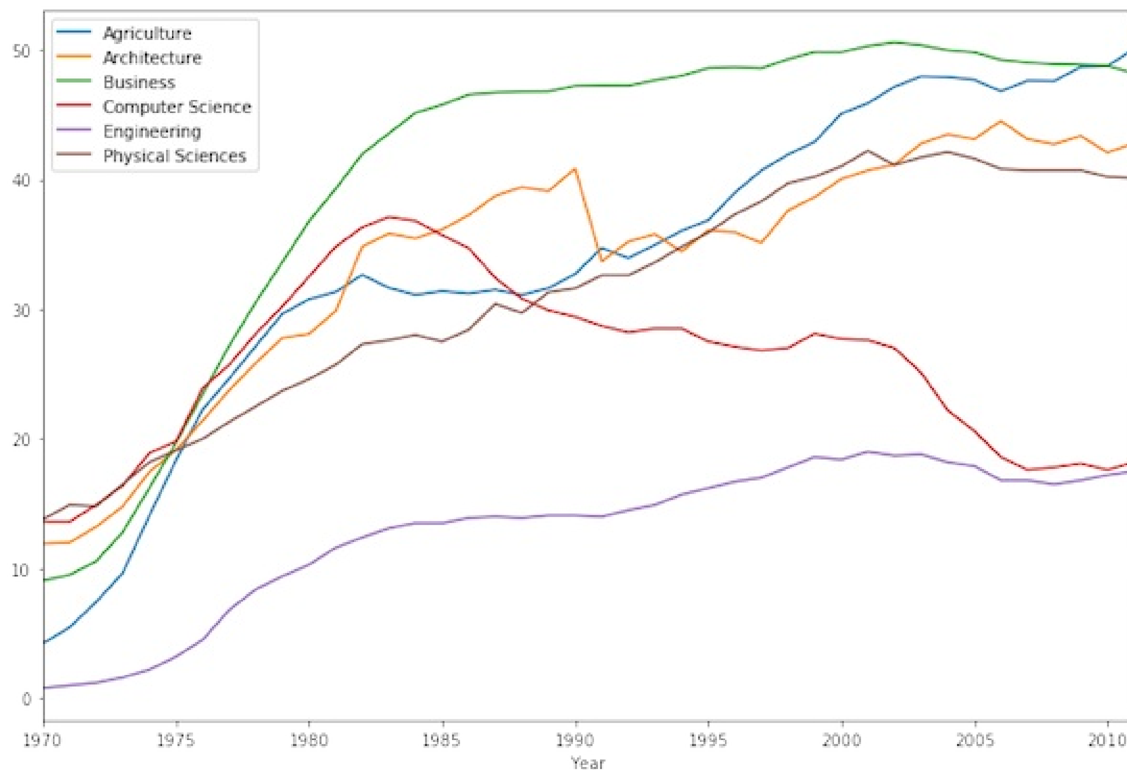
By a similar derivation we can also show that if the relationship between the $ \log(y) $ and $ x $ is linear, the two variables have an exponential relationship: $ y = a^x $.

Thus, we can use the logarithm to reveal patterns in right-tailed data and common non-linear relationships between variables.

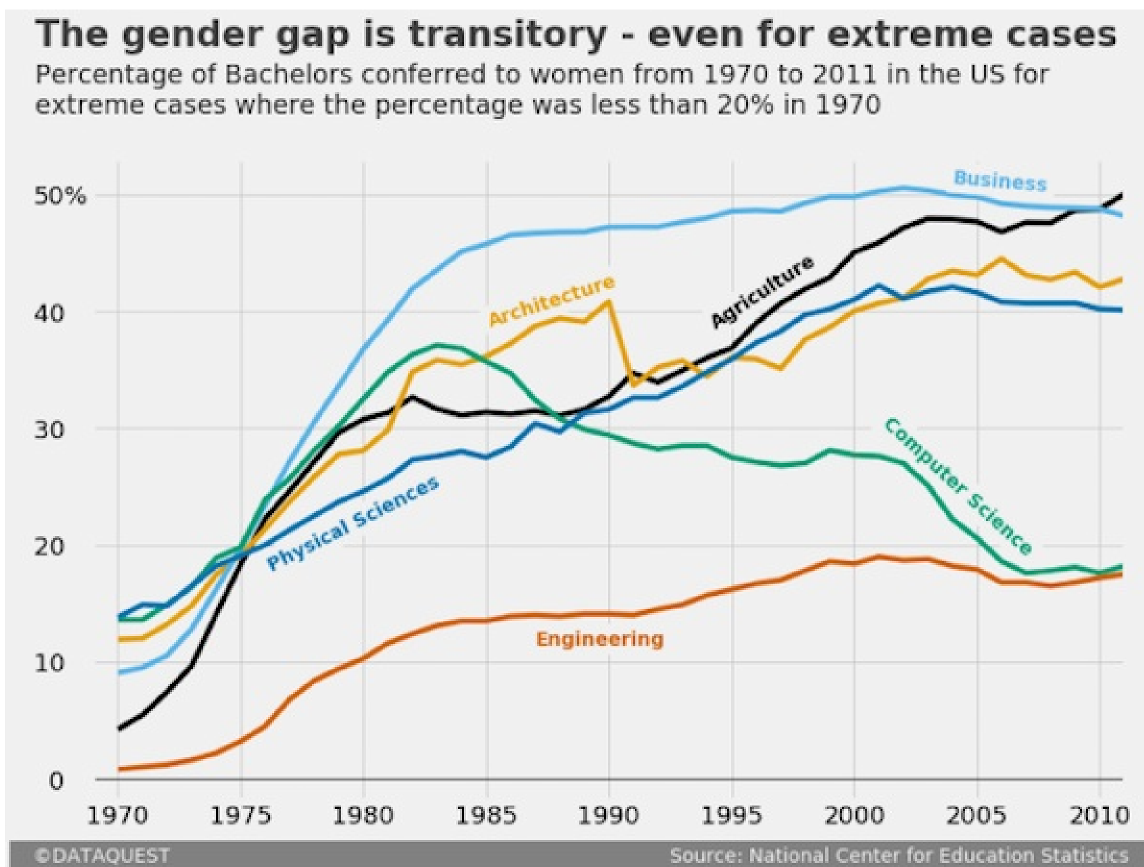
Other common data transformations include the Box-Cox transformation and polynomial transforms.

Principles of Context¶

It is important to add as much relevant context as possible to any plot you plan to share more broadly. For example, the following plot shows its data clearly but provides little context to help understand what is being plotted.



To provide context, we add a title, caption, axes labels, units for the axes, and labels for the plotted lines.



(This blog post explains how to make these modifications using `matplotlib`.)

In general, we provide context for a plot through:

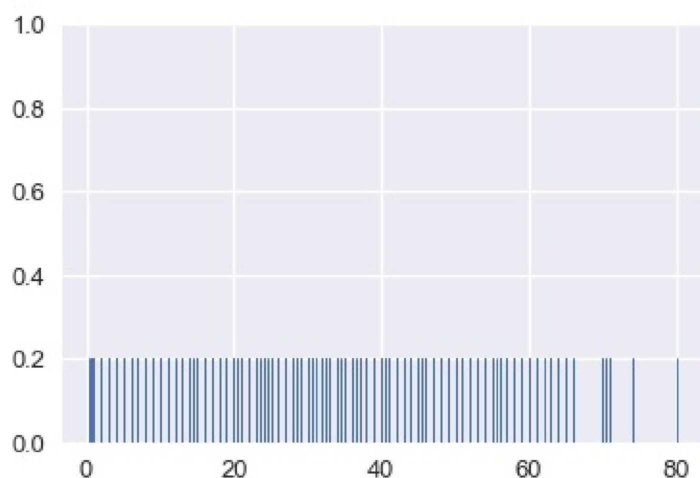
- Plot title
- Axes labels
- Reference lines and markers for important values
- Labels for interesting points and unusual observations
- Captions that describe the data and its important features

Principles of Smoothing¶

Smoothing allows us to more clearly visualize data when we have many data points. We've actually already seen an instance of smoothing: histograms are a type of smoothing for rugplots. This rugplot shows each age of the passengers in the Titanic.

```
ages = ti['age'].dropna()
sns.rugplot(ages, height=0.2)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a20c05b38>
```

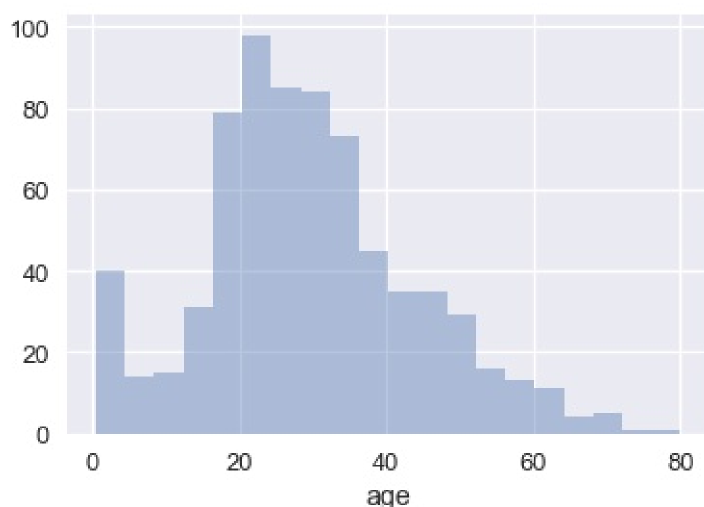


There are many marks that make it difficult to tell where the data lie. In addition, some of the points overlap, making it impossible to see how many points lie at 0. This issue is called *overplotting* and we generally avoid it whenever possible.

To reveal the distribution of the data, we can replace groups of marks with a bar that is taller when more points are in the group. Smoothing refers to this process of replacing sets of points with appropriate markers; we choose not to show every single point in the dataset in order to reveal broader trends.

```
sns.distplot(ages, kde=False)
```

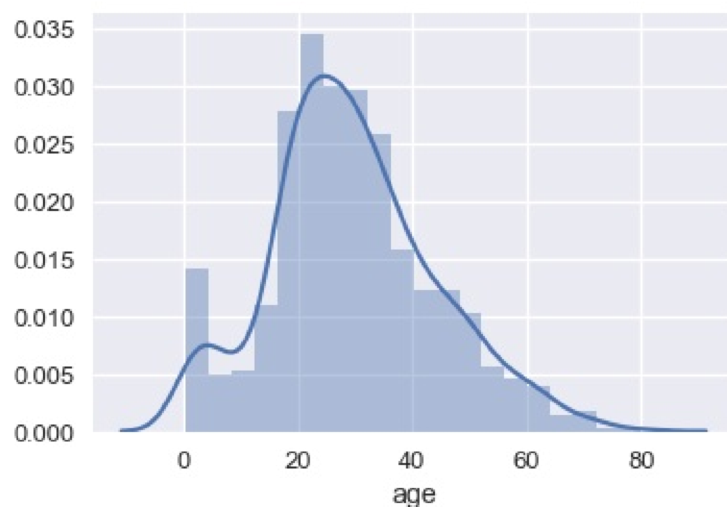
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a23c384e0>
```



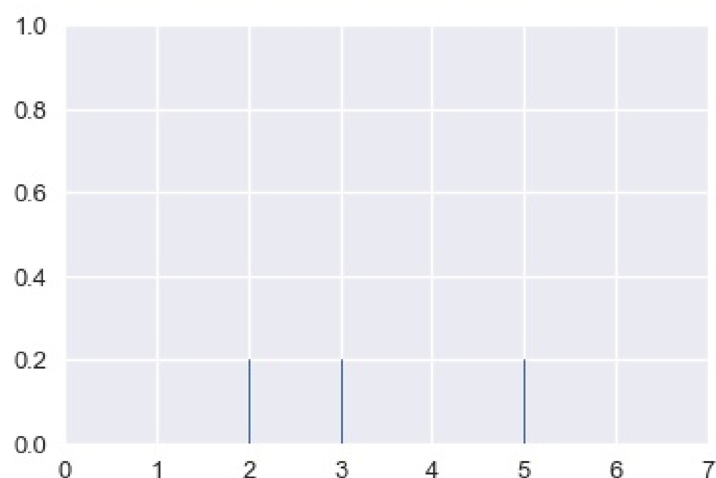
We've also seen that `seaborn` will plot a smooth curve over a histogram by default.

```
sns.distplot(ages)
```

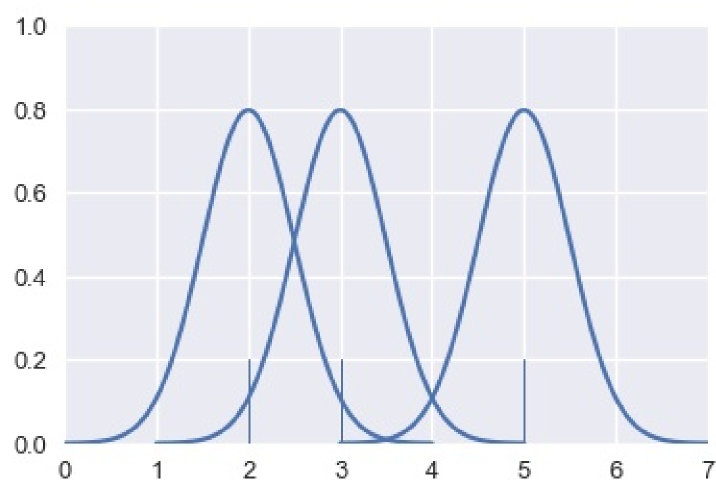
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a23d89780>
```



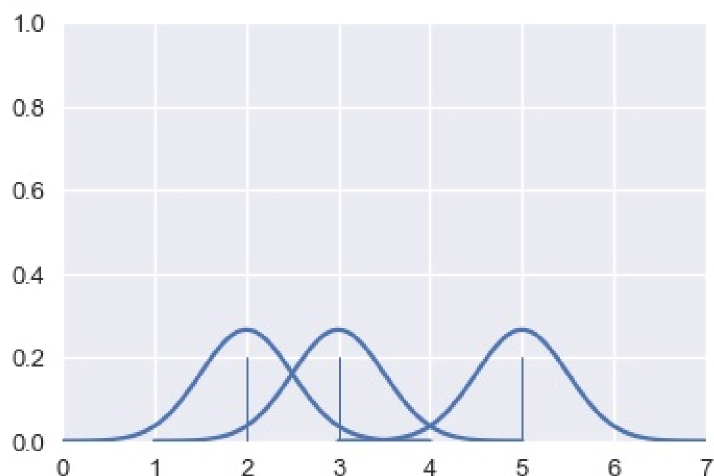
This is another form of smoothing called *kernel density estimation* (KDE). Instead of grouping points together and plotting bars, KDE places a curve on each point and combines the individual curves to create a final estimation of the distribution. Consider the rugplot below that shows three points.



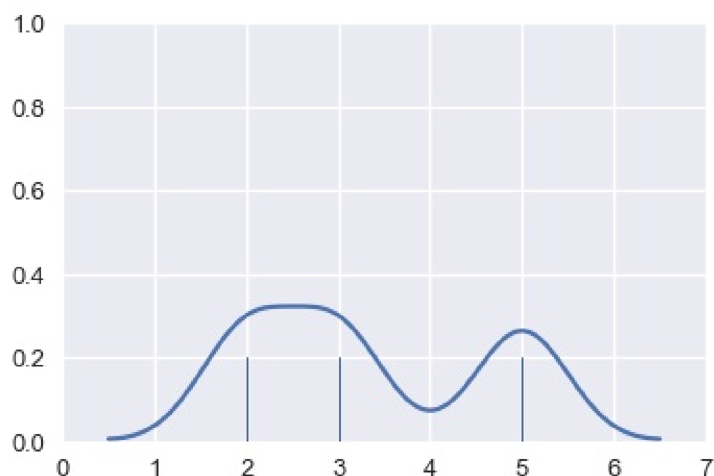
To perform KDE, we place a Gaussian (normal) distribution on each point:



The area under each Gaussian curve is equal to 1. Since we will sum multiple curves together, we scale each curve so that when added together the area under all the curves is equal to 1.



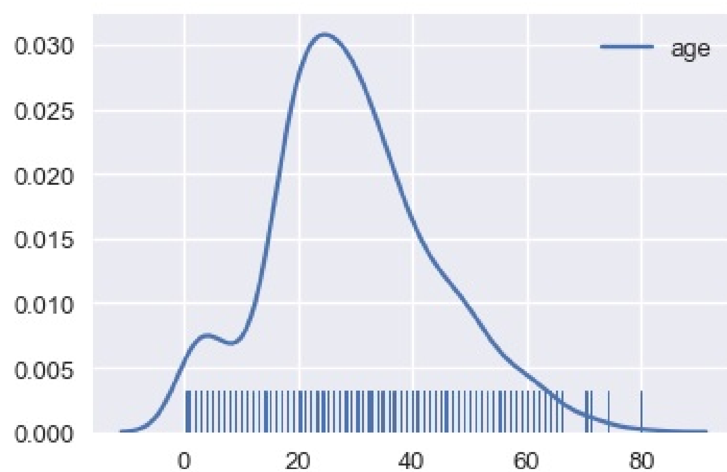
Finally, we add the curves together to create a final smooth estimate for the distribution:



By following this procedure, we can use KDE to smooth many points.

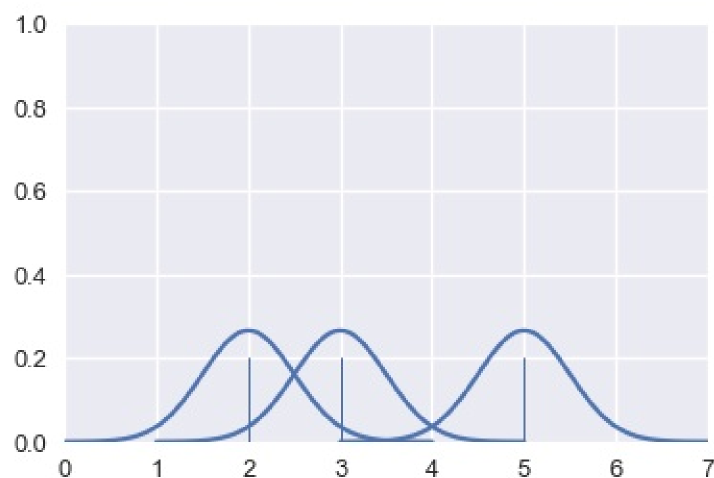
```
# Show the original unsmoothed points
sns.rugplot(ages, height=0.1)

# Show the smooth estimation of the distribution
sns.kdeplot(ages);
```

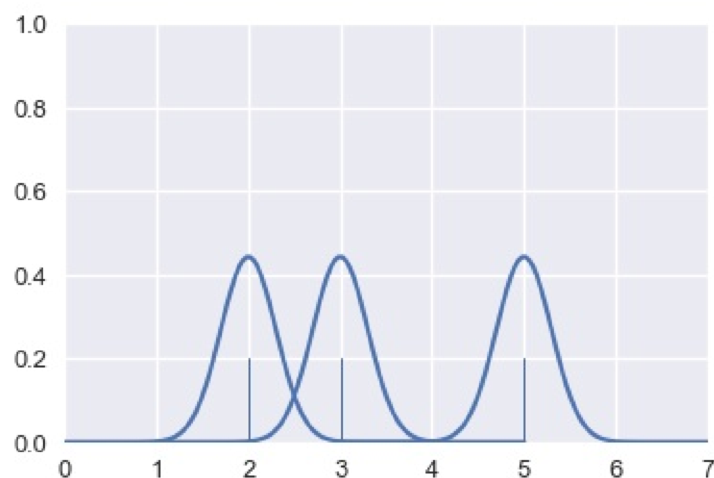


Kernel Density Estimation Details¶

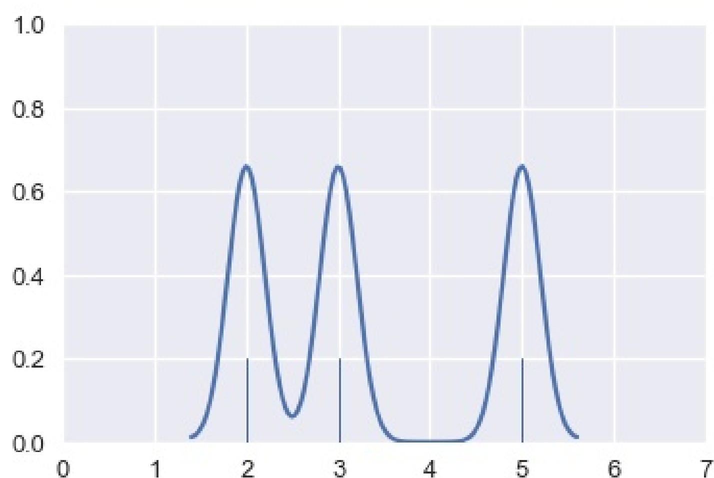
In the previous examples of KDE, we placed a miniature Gaussian curve on each point and added the Gaussians together.



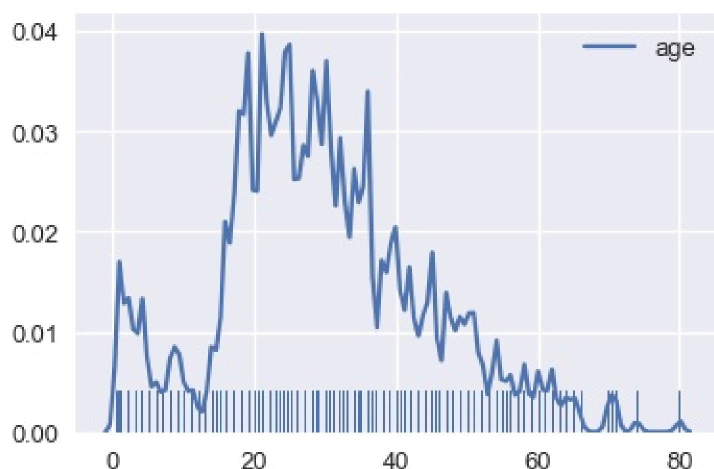
We are free to adjust the width of the Gaussians. For example, we can make each Gaussian narrower. This is called decreasing the *bandwidth* of the kernel estimation.



When we add these narrower Gaussians together, we create a more detailed final estimation.



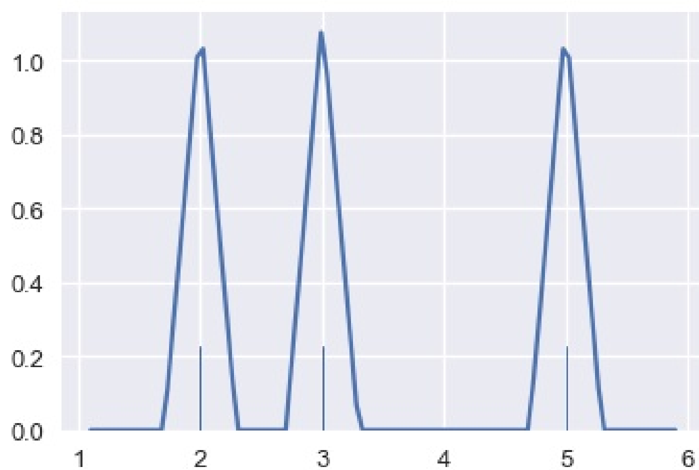

```
# Plot the KDE for Titanic passenger ages using a lower
bandwidth
sns.rugplot(ages, height=0.1)
sns.kdeplot(ages, bw=0.5);
```



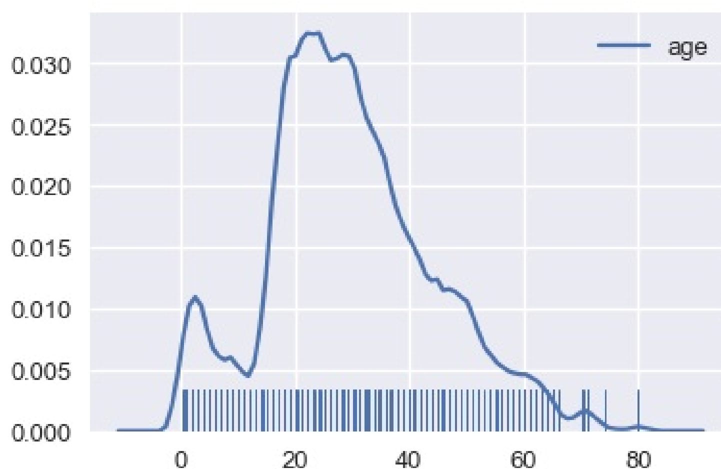
Just like adjusting bins for a histogram, we typically adjust the bandwidth until we believe the final plot shows the distribution without distracting the viewer with too much detail.

Although we have placed a Gaussian at each point so far, we can easily select other functions to estimate each point. This is called changing the *kernel* of the kernel density estimation. Previously, we've used a Gaussian kernel. Now, we'll use a triangular kernel which places a pair of stepwise sloped lines at each point:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a26eb3908>
```




```
# Plot the KDE for Titanic passenger ages using a triangular
kernel
sns.rugplot(ages, height=0.1)
sns.kdeplot(ages, kernel='tri');
```



Usually we'll use a Gaussian kernel unless we have a specific reason to use a different kernel.

Smoothing a Scatter Plot¶

We can also smooth two-dimensional plots when we encounter the problem of overplotting.

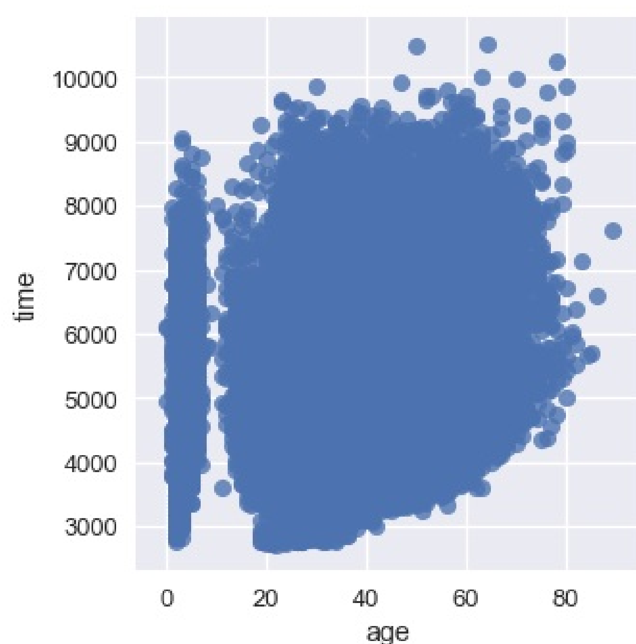
The following example comes from a dataset released by the Cherry Blossom Run, an annual 10-mile run in Washington D.C. Each runner can report their age and their race time; we've plotted all the reported data points in the scatter plot below.

```
runners = pd.read_csv('data/cherryBlossomMen.csv').dropna()
runners
```

	year	place	age	time
0	1999	1	28.0	2819.0
1	1999	2	24.0	2821.0
2	1999	3	27.0	2823.0
...	...	...	...	...
70066	2012	7190	56.0	8840.0
70067	2012	7191	35.0	8850.0
70069	2012	7193	48.0	9059.0

70045 rows × 4 columns

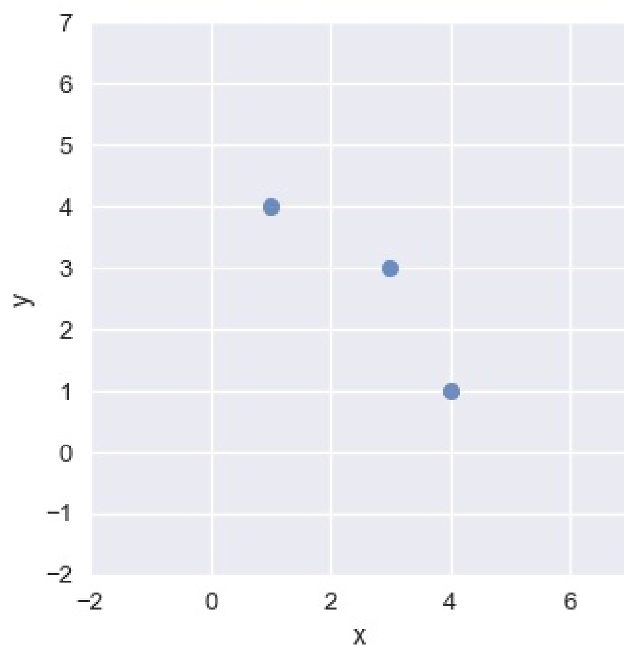
```
sns.lmplot(x='age', y='time', data=runners, fit_reg=False);
```



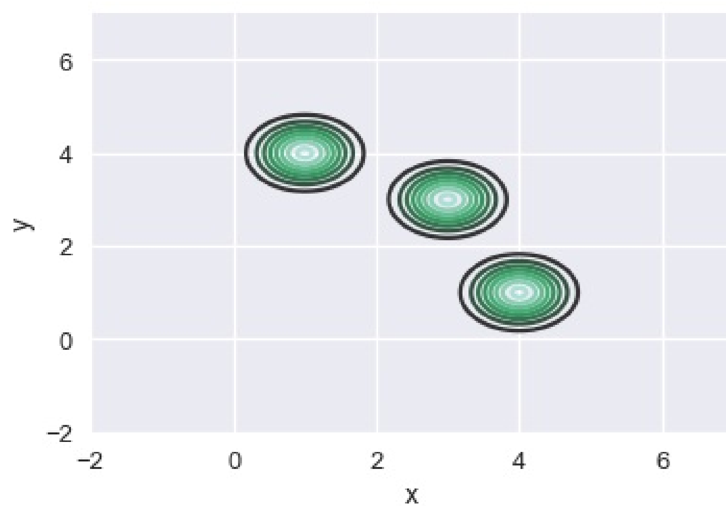
So many points lie on top of each other that it's difficult to see any trend at all!

We can smooth the scatter plot using kernel density estimation in two dimensions. When KDE is applied to a two-dimensional plot, we place a three-dimensional Gaussian at each point. In three dimensions, the Gaussian looks like a mountain pointing out of the page.

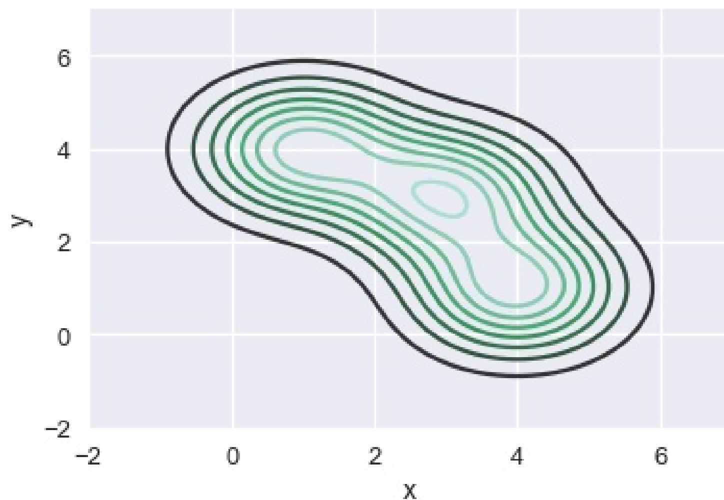
```
# Plot three points
two_d_points = pd.DataFrame({'x': [1, 3, 4], 'y': [4, 3, 1]})
sns.lmplot(x='x', y='y', data=two_d_points, fit_reg=False)
plt.xlim(-2, 7)
plt.ylim(-2, 7);
```




```
# Place a Gaussian at each point and use a contour plot to show
each one
sns.kdeplot(two_d_points['x'], two_d_points['y'], bw=0.4)
plt.xlim(-2, 7)
plt.ylim(-2, 7);
```

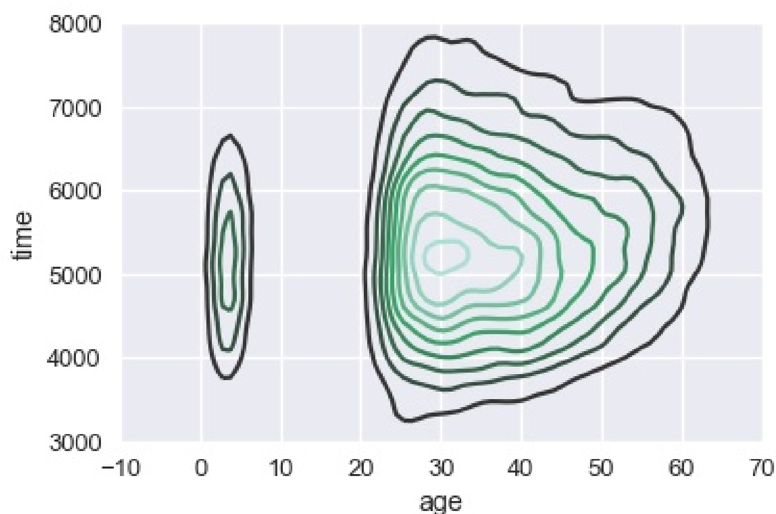


Just like we've previously seen, we scale each Gaussian and add them together to obtain a final contour plot for the scatter plot.



The resulting plot shows the downward sloping trend of the three points. Similarly, we can apply a KDE to smooth out the scatter plot of runner ages and times:

```
sns.kdeplot(runners['age'], runners['time'])
plt.xlim(-10, 70)
plt.ylim(3000, 8000);
```



We can see that most of our runners were between 25 and 50 years old, and that most runners took between 4000 and 7000 seconds (roughly between 1 and 2 hours) to finish the race.

We can see more clearly that there is a suspicious group of runners that are between zero and ten years old. We might want to double check that our data for those ages was recorded properly.

We can also see a slight upward trend in the time taken to finish the race as runner age increases.

Philosophy for Data Visualization

Unfortunately, many charts in the world at large do not adhere to our principles for data visualization. Here is a page of charts that appear on the first page of a Google search for "good data visualization" conducted in Spring of 2018. How many poorly made charts can you spot?

Effective data visualization reveals the data. All of our principles for visualization aim to make the data more understandable for the viewer. Data visualization is sometimes used to mislead and misinform. When done properly, however, data visualization is one of our greatest tools for discovering, revealing, and communicating trends and anomalies in our data.

Web Technologies

Before the Internet, data scientists had to physically move hard disk drives to share data with others. Now, we can freely retrieve datasets from computers across the world.

Although we use the Internet to download and share data files, the web pages on the Internet themselves contain huge amounts of information as text, images, and videos. By learning web technologies, we can use the Web as a data source. In this chapter, we introduce HTTP, the primary communication protocol for the Web, and XML/HTML, the primary document formats for web pages.

Show Widgets Open on DataHub

Table of Contents

HTTP¶

HTTP (AKA **H**yper**T**ext **T**ransfer **P**rotocol) is a *request-response* protocol that allows one computer to talk to another over the Internet.

Requests and Responses¶

The Internet allows computers to send text to one another, but does not impose any restrictions on what that text contains. HTTP defines a structure on the text communication between one computer (client) and another (server). In this protocol, a client submits a *request* to a server, a specially formatted text message. The server sends a text *response* back to the client.

The command line tool `curl` gives us a simple way to send HTTP requests. In the output below, lines starting with `>` indicate the text sent in our request; the remaining lines are the server's response.

```
$ curl -v https://httpbin.org/html
```

```
> GET /html HTTP/1.1
> Host: httpbin.org
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Server: meinheld/0.6.1
< Date: Wed, 11 Apr 2018 18:15:03 GMT
<
<html>
  <body>
    <h1>Herman Melville - Moby-Dick</h1>
    <p>
      Availing himself of the mild...
    </p>
  </body>
</html>
```

Running the `curl` command above causes the client's computer to construct a text message that looks like:

```
GET /html HTTP/1.1
Host: httpbin.org
User-Agent: curl/7.55.1
Accept: */*
{blank_line}
```

This message follows a specific format: it starts with `GET /html HTTP/1.1` which indicates that the message is an HTTP `GET` request to the `/html` page. Each of the three lines that follow form HTTP headers, optional information that `curl` sends to the server. The HTTP headers have the format `{name}: {value}`. Finally, the blank line at the end of the message tells the server that the message ends after three headers. Note that we've marked the blank line with `{blank_line}` in the snippet above; in the actual message `{blank_line}` is replaced with a blank line.

The client's computer then uses the Internet to send this message to the `https://httpbin.org` web server. The server processes the request, and sends the following response:

```
HTTP/1.1 200 OK
Connection: keep-alive
Server: meinheld/0.6.1
Date: Wed, 11 Apr 2018 18:15:03 GMT
{blank_line}
```

The first line of the response states that the request completed successfully. The following three lines form the HTTP response headers, optional information that the server sends back to the client. Finally, the blank line at the end of the message tells the client that the server has finished sending its response headers and will next send the response body:

```
<html>
  <body>
    <h1>Herman Melville - Moby-Dick</h1>
    <p>
      Availing himself of the mild...
    </p>
  </body>
</html>
```

This HTTP protocol is used in almost every application that interacts with the Internet. For example, visiting https://httpbin.org/html in your web browser makes the same basic HTTP request as the `curl` command above. Instead of displaying the response as plain text as we have above, your browser recognizes that the text is an HTML document and will display it accordingly.

In practice, we will not write out full HTTP requests in text. Instead, we use tools like `curl` or Python libraries to construct requests for us.

In Python¶

The Python **requests** library allows us to make HTTP requests in Python. The code below makes the same HTTP request as running `curl -v https://httpbin.org/html`.

```
import requests

url = "https://httpbin.org/html"
response = requests.get(url)
response
```

```
<Response [200]>
```

The Request¶

Let's take a closer look at the request we made. We can access the original request using `response` object; we display the request's HTTP headers below:

```
request = response.request
for key in request.headers: # The headers in the response are stored as a dictionary.
    print(f'{key}: {request.headers[key]}')
```

```
User-Agent: python-requests/2.12.4
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
```

Every HTTP request has a type. In this case, we used a `GET` request which retrieves information from a server.

```
request.method
```

```
'GET'
```

The Response¶

Let's examine the response we received from the server. First, we will print the response's HTTP headers.

```
for key in response.headers:
    print(f'{key}: {response.headers[key]}')
```

```
Connection: keep-alive
Server: gunicorn/19.7.1
Date: Wed, 25 Apr 2018 18:32:51 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 3741
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
X-Powered-By: Flask
X-Processed-Time: 0
Via: 1.1 vegur
```

An HTTP response contains a status code, a special number that indicates whether the request succeeded or failed. The status code `200` indicates that the request succeeded.

```
response.status_code
```

```
200
```

Finally, we display the first 100 characters of the response's content (the entire response content is too long to display nicely here).

```
response.text[:100]
```

```
'<!DOCTYPE html>\n<html>\n  <head>\n  </head>\n  <body>\n
<h1>Herman Melville - Moby-Dick</h1>\n\n     '
```

Types of Requests¶

The request we made above was a `GET` HTTP request. There are multiple HTTP request types; the most important two are `GET` and `POST` requests.

GET Requests¶

The `GET` request is used to retrieve information from the server. Since your web browser makes `GET` request whenever you enter in a URL into its address bar, `GET` requests are the most common type of HTTP requests.

`curl` uses `GET` requests by default, so running `curl https://www.google.com/` makes a `GET` request to `https://www.google.com/` .

POST Request¶

The `POST` request is used to send information from the client to the server. For example, some web pages contain forms for the user to fill out—a login form, for example. After clicking the "Submit" button, most web browsers will make a `POST` request to send the form data to the server for processing.

Let's look an example of a `POST` request that sends `'sam'` as the parameter `'name'`. This one can be done by running **`curl -d 'name=sam' https://httpbin.org/post`** on the command line.

Notice that our request has a body this time (filled with the parameters of the `POST` request), and the content of the response is different from our `GET` response from before.

Like HTTP headers, the data sent in a `POST` request uses a key-value format. In Python, we can make a `POST` request by using `requests.post` and passing in a dictionary as an argument.

```
post_response = requests.post("https://httpbin.org/post",
                              data={'name': 'sam'})
post_response
```

```
<Response [200]>
```

The server will respond with a status code to indicate whether the `POST` request successfully completed. In addition, the server will usually send a response body to display to the client.

```
post_response.status_code
```

```
200
```

```
post_response.text
```

```
'{\n  "args": {}, \n  "data": "", \n  "files": {}, \n  "form": 
{\n    "name": "sam"\n  }, \n  "headers": {\n    "Accept": 
"*/*", \n    "Accept-Encoding": "gzip, deflate", \n    
"Connection": "close", \n    "Content-Length": "8", \n    
"Content-Type": "application/x-www-form-urlencoded", \n    
"Host": "httpbin.org", \n    "User-Agent": "python-
requests/2.12.4"\n  }, \n  "json": null, \n  "origin": 
"136.152.143.72", \n  "url": "https://httpbin.org/post"\n}\n'
```

Types of Response Status Codes¶

The previous HTTP responses had the HTTP status code `200`. This status code indicates that the request completed successfully. There are hundreds of other HTTP status codes. Thankfully, they are grouped into categories to make them easier to remember:

- **100s** - Informational: More input is expected from client or server *(e.g. 100 Continue, 102 Processing)*
- **200s** - Success: The client's request was successful *(e.g. 200 OK, 202 Accepted)*
- **300s** - Redirection: Requested URL is located elsewhere; May need user's further action *(e.g. 300 Multiple Choices, 301 Moved Permanently)*
- **400s** - Client Error: Client-side error *(e.g. 400 Bad Request, 403 Forbidden, 404 Not Found)*
- **500s** - Server Error: Server-side error or server is incapable of performing the request *(e.g. 500 Internal Server Error, 503 Service Unavailable)*

We can look at examples of some of these errors.

```
# This page doesn't exist, so we get a 404 page not found error
url = "https://www.youtube.com/404errorwow"
errorResponse = requests.get(url)
print(errorResponse)
```

```
<Response [404]>
```

```
# This specific page results in a 500 server error
url = "https://httpstat.us/500"
serverResponse = requests.get(url)
print(serverResponse)
```

```
<Response [500]>
```

Summary¶

We have introduced the HTTP protocol, the basic communication method for applications that use the Web. Although the protocol specifies a specific text format, we typically turn to other tools to make HTTP requests for us, such as the command line tool `curl` and the

Python library `requests`.

Working with Text

A great quantity of data resides not as numbers in CSVs but as free-form text in books, documents, blog posts, and Internet comments. While numerical and categorical data are often collected from physical phenomena, textual data arises from human communication and expression. As with most types of data, there are a multitude of techniques for working with text that would take multiple books to explain in full detail. In this chapter, we introduce a small subset of these techniques that provide a variety of useful operations for working with text: Python string manipulation and regular expressions.

Show Widgets Open on DataHub

Table of Contents

Python String Methods¶

Python provides a variety of methods for basic string manipulation. Although simple, these methods form the primitives that piece together to form more complex string operations. We will introduce Python's string methods in the context of a common use case for working with text: data cleaning.

Cleaning Text Data¶

Data often comes from several different sources that each implements its own way of encoding information. In the following example, we have one table that records the state that a county belongs to and another that records the population of the county.

```
state
```

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LA

```
population
```

	County	Population
0	DeWitt	16,798
1	Lac Qui Parle	8,067
2	Lewis & Clark	55,716
3	St. John the Baptist	43,044

We would naturally like to join the `state` and `population` tables using the `County` column. Unfortunately, not a single county is spelled the same in the two tables. This example is illustrative of the following common issues in text data:

1. Capitalization: `qui` vs `Qui`
2. Different punctuation conventions: `St.` vs `St`
3. Omission of words: `County` / `Parish` is absent in the `population` table
4. Use of whitespace: `DeWitt` vs `De Witt`
5. Different abbreviation conventions: `&` vs `and`

String Methods¶

Python's string methods allow us to start resolving these issues. These methods are conveniently defined on all Python strings and thus do not require importing other modules. Although it is worth familiarizing yourself with the complete list of string methods, we describe a few of the most commonly used methods in the table below.

Method	Description
`str[x:y]`	Slices `str`, returning indices x (inclusive) to y (not inclusive)
`str.lower()`	Returns a copy of a string with all letters converted to lowercase
`str.replace(a, b)`	Replaces all instances of the substring `a` in `str` with the substring `b`
`str.split(a)`	Returns substrings of `str` split at a substring `a`
`str.strip()`	Removes leading and trailing whitespace from `str`

We select the string for St. John the Baptist parish from the `state` and `population` tables and apply string methods to remove capitalization, punctuation, and `county` / `parish` occurrences.

```
john1 = state.loc[3, 'County']
john2 = population.loc[0, 'County']

(john1
 .lower()
 .strip()
 .replace(' parish', '')
 .replace(' county', '')
 .replace('&', 'and')
 .replace('.', '')
 .replace(' ', '')
)
```

```
'stjohnthebaptist'
```

Applying the same set of methods to `john2` allows us to verify that the two strings are now identical.

```
(john1
 .lower()
 .strip()
 .replace(' parish', '')
 .replace(' county', '')
 .replace('&', 'and')
 .replace('.', '')
 .replace(' ', '')
)
```

```
'stjohnthebaptist'
```

Satisfied, we create a method called `clean_county` that normalizes an input county.

```
def clean_county(county):
    return (county
            .lower()
            .strip()
            .replace(' county', '')
            .replace(' parish', '')
            .replace('&', 'and')
            .replace(' ', '')
            .replace('.', ''))
```

We may now verify that the `clean_county` method produces matching counties for all the counties in both tables:

```
([clean_county(county) for county in state['County']],
 [clean_county(county) for county in population['County']]
)
```

```
(['dewitt', 'lacquiparle', 'lewisandclark', 'stjohnthebaptist'],
 ['dewitt', 'lacquiparle', 'lewisandclark', 'stjohnthebaptist'])
```

Because each county in both tables has the same transformed representation, we may successfully join the two tables using the transformed county names.

String Methods in pandas¶

In the code above we used a loop to transform each county name. `pandas` Series objects provide a convenient way to apply string methods to each item in the series. First, the series of county names in the `state` table:

```
state['County']
```

```
0                 De Witt County
1           Lac qui Parle County
2             Lewis and Clark County
3      St John the Baptist Parish
Name: County, dtype: object
```

The `.str` property on `pandas` Series exposes the same string methods as Python does. Calling a method on the `.str` property calls the method on each item in the series.

```
state['County'].str.lower()
```

```
0                 de witt county
1           lac qui parle county
2         lewis and clark county
3    st john the baptist parish
Name: County, dtype: object
```

This allows us to transform each string in the series without using a loop.

```
(state['County']
 .str.lower()
 .str.strip()
 .str.replace(' parish', '')
 .str.replace(' county', '')
 .str.replace('&', 'and')
 .str.replace('.', '')
 .str.replace(' ', '')
)
```

```
0               dewitt
1          lacquiparle
2        lewisandclark
3    stjohnthebaptist
Name: County, dtype: object
```

We save the transformed counties back into their originating tables:

```
state['County'] = (state['County']
 .str.lower()
 .str.strip()
 .str.replace(' parish', '')
 .str.replace(' county', '')
 .str.replace('&', 'and')
 .str.replace('.', '')
 .str.replace(' ', '')
)

population['County'] = (population['County']
 .str.lower()
 .str.strip()
 .str.replace(' parish', '')
 .str.replace(' county', '')
 .str.replace('&', 'and')
 .str.replace('.', '')
 .str.replace(' ', '')
)
```

Now, the two tables contain the same string representation of the counties:

```
state
```

	County	State
0	dewitt	IL
1	lacquiparle	MN
2	lewisandclark	MT
3	stjohnthebaptist	LA

```
population
```

	County	Population
0	dewitt	16,798
1	lacquiparle	8,067
2	lewisandclark	55,716
3	stjohnthebaptist	43,044

It is simple to join these tables once the counties match.

```
state.merge(population, on='County')
```

	County	State	Population
0	dewitt	IL	16,798
1	lacquiparle	MN	8,067
2	lewisandclark	MT	55,716
3	stjohnthebaptist	LA	43,044

Summary¶

Python's string methods form a set of simple and useful operations for string manipulation. `pandas` Series implement the same methods that apply the underlying Python method to each string in the series.

You may find the complete docs on Python's `string` methods here and the docs on Pandas `str` methods here.

Show Widgets Open on DataHub

Table of Contents

Regular Expressions¶

In this section we introduce regular expressions, an important tool to specify patterns in strings.

Motivation¶

In a larger piece of text, many useful substrings come in a specific format. For instance, the sentence below contains a U.S. phone number.

`"give me a call, my number is 123-456-7890."`

The phone number contains the following pattern:

1. Three numbers
2. Followed by a dash
3. Followed by three numbers
4. Followed by a dash
5. Followed by four Numbers

Given a free-form segment of text, we might naturally wish to detect and extract the phone numbers. We may also wish to extract specific pieces of the phone numbers—for example, by extracting the area code we may deduce the locations of individuals mentioned in the

text.

To detect whether a string contains a phone number, we may attempt to write a method like the following:

```
def is_phone_number(string):

    digits = '0123456789'

    def is_not_digit(token):
        return token not in digits

    # Three numbers
    for i in range(3):
        if is_not_digit(string[i]):
            return False

    # Followed by a dash
    if string[3] != '-':
        return False

    # Followed by three numbers
    for i in range(4, 7):
        if is_not_digit(string[i]):
            return False

    # Followed by a dash
    if string[7] != '-':
        return False

    # Followed by four numbers
    for i in range(8, 12):
        if is_not_digit(string[i]):
            return False

    return True
```

```
is_phone_number("382-384-3840")
```

```
True
```

```
is_phone_number("phone number")
```

```
False
```

The code above is unpleasant and verbose. Rather than manually loop through the characters of the string, we would prefer to specify a pattern and command Python to match the pattern.

Regular expressions (often abbreviated **regex**) conveniently solve this exact problem by allowing us to create general patterns for strings. Using a regular expression, we may re-implement the `is_phone_number` method in two short lines of Python:

```
import re

def is_phone_number(string):
    regex = r"[0-9]{3}-[0-9]{3}-[0-9]{4}"
    return re.search(regex, string) is not None

is_phone_number("382-384-3840")
```

```
True
```

In the code above, we use the regex `[0-9]{3}-[0-9]{3}-[0-9]{4}` to match phone numbers. Although cryptic at a first glance, the syntax of regular expressions is fortunately much simpler to learn than the Python language itself; we introduce nearly all of the syntax in this section alone.

We will also introduce the built-in Python module `re` that performs string operations using regexes.

Regex Syntax¶

We start with the syntax of regular expressions. In Python, regular expressions are most commonly stored as raw strings. Raw strings behave like normal Python strings without special handling for backslashes.

For example, to store the string `hello \ world` in a normal Python string, we must write:

```
# Backslashes need to be escaped in normal Python strings
some_string = 'hello \\ world'
print(some_string)
```

```
hello \ world
```

Using a raw string removes the need to escape the backslash:

```
# Note the `r` prefix on the string
some_raw_string = r'hello \ world'
print(some_raw_string)
```

```
hello \ world
```

Since backslashes appear often in regular expressions, we will use raw strings for all regexes in this section.

Literals¶

A **literal** character in a regular expression matches the character itself. For example, the regex `r"a"` will match any `"a"` in `"Say! I like green eggs and ham!"`. All alphanumeric characters and most punctuation characters are regex literals.

```
# The show_regex_match method highlights all regex matches in
the input string
regex = r"green"
show_regex_match("Say! I like green eggs and ham!", regex)
```

```
Say! I like green eggs and ham!
```

```
show_regex_match("Say! I like green eggs and ham!", r"a")
```

```
Say! I like green eggs and ham!
```

In the example above we observe that regular expressions can match patterns that appear anywhere in the input string. In Python, this behavior differs depending on the method used to match the regex—some methods only return a match if the regex appears at the start of the string; some methods return a match anywhere in the string.

Notice also that the `show_regex_match` method highlights all occurrences of the regex in the input string. Again, this differs depending on the Python method used—some methods return all matches while some only return the first match.

Regular expressions are case-sensitive. In the example below, the regex only matches the lowercase `s` in `eggs`, not the uppercase `S` in `Say`.

```
show_regex_match("Say! I like green eggs and ham!", r"s")
```

```
Say! I like green eggs and ham!
```

Wildcard Character¶

Some characters have special meaning in a regular expression. These meta characters allow regexes to match a variety of patterns.

In a regular expression, the period character `.` matches any character except a newline.

```
show_regex_match("Call me at 382-384-3840.", r".all")
```

```
Call me at 382-384-3840.
```

To match only the literal period character we must escape it with a backslash:

```
show_regex_match("Call me at 382-384-3840.", r"\.")
```

```
Call me at 382-384-3840.
```

By using the period character to mark the parts of a pattern that vary, we construct a regex to match phone numbers. For example, we may take our original phone number `382-384-3840` and replace the numbers with `.`, leaving the dashes as literals. This results in the regex `...-...-....`.

```
show_regex_match("Call me at 382-384-3840.", "...-...-....")
```

```
Call me at 382-384-3840.
```

Since the period character matches all characters, however, the following input string will produce a spurious match.

```
show_regex_match("My truck is not-all-blue.", "...-...-....")
```

```
My truck is not-all-blue.
```

Character Classes¶

A **character class** matches a specified set of characters, allowing us to create more restrictive matches than the `.` character alone. To create a character class, wrap the set of desired characters in brackets `[ ]`.

```
show_regex_match("I like your gray shirt.", "gr[ae]y")
```

```
I like your gray shirt.
```

```
show_regex_match("I like your grey shirt.", "gr[ae]y")
```

```
I like your grey shirt.
```

```
# Does not match; a character class only matches one character
from a set
show_regex_match("I like your graey shirt.", "gr[ae]y")
```

```
I like your graey shirt.
```

```
# In this example, repeating the character class will match
show_regex_match("I like your graey shirt.", "gr[ae][ae]y")
```

```
I like your graey shirt.
```

In a character class, the `.` character is treated as a literal, not as a wildcard.

```
show_regex_match("I like your grey shirt.", "irt[.]")
```

```
I like your grey shirt.
```

There are a few special shorthand notations we can use for commonly used character classes:

Shorthand	Meaning
[0-9]	All the digits
[a-z]	Lowercase letters
[A-Z]	Uppercase letters

```
show_regex_match("I like your gray shirt.", "y[a-z]y")
```

```
I like your gray shirt.
```

Character classes allow us to create a more specific regex for phone numbers.

```
# We replaced every `.` character in ...-...-.... with [0-9] to
restrict
# matches to digits.
phone_regex = r'[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9]
[0-9]'
show_regex_match("Call me at 382-384-3840.", phone_regex)
```

```
Call me at 382-384-3840.
```

```
# Now we no longer match this string:
show_regex_match("My truck is not-all-blue.", phone_regex)
```

```
My truck is not-all-blue.
```

Negated Character Classes¶

A **negated character class** matches any character **except** the characters in the class. To create a negated character class, wrap the negated characters in `[^ ]`.

```
show_regex_match("The car parked in the garage.", r"[^c]ar")
```

```
The car parked in the garage.
```

Quantifiers¶

To create a regex to match phone numbers, we wrote:

```
[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
```

This matches 3 digits, a dash, 3 more digits, a dash, and 4 more digits.

Quantifiers allow us to match multiple consecutive appearances of a pattern. We specify the number of repetitions by placing the number in curly braces `{ }`.

```
phone_regex = r'[0-9]{3}-[0-9]{3}-[0-9]{4}'
show_regex_match("Call me at 382-384-3840.", phone_regex)
```

```
Call me at 382-384-3840.
```

```
# No match
phone_regex = r'[0-9]{3}-[0-9]{3}-[0-9]{4}'
show_regex_match("Call me at 12-384-3840.", phone_regex)
```

```
Call me at 12-384-3840.
```

A quantifier always modifies the character or character class to its immediate left. The following table shows the complete syntax for quantifiers.

Quantifier	Meaning
{m, n}	Match the preceding character m to n times.
{m}	Match the preceding character exactly m times.
{m,}	Match the preceding character at least m times.
{,n}	Match the preceding character at most n times.

Shorthand Quantifiers

Some commonly used quantifiers have a shorthand:

Symbol	Quantifier	Meaning
*	{0,}	Match the preceding character 0 or more times
+	{1,}	Match the preceding character 1 or more times
?	{0,1}	Match the preceding charcter 0 or 1 times

We use the `*` character instead of `{0,}` in the following examples.

```
# 3 a's
show_regex_match('He screamed "Aaaah!" as the cart took a
plunge.', "Aa*h!")
```

```
He screamed "Aaaah!" as the cart took a plunge.
```

```
# Lots of a's
show_regex_match(
    'He screamed "Aaaaaaaaaaaaaaaaaaaah!" as the cart took a
plunge.',
    "Aa*h!"
)
```

```
He screamed "Aaaaaaaaaaaaaaaaaaaah!" as the cart took a plunge.
```

```
# No lowercase a's
show_regex_match('He screamed "Ah!" as the cart took a plunge.',
"Aa*h!")
```

```
He screamed "Ah!" as the cart took a plunge.
```

Quantifiers are greedy

Quantifiers will always return the longest match possible. This sometimes results in surprising behavior:

```
# We tried to match 311 and 911 but matched the ` and ` as well
because
# `<311> and <911>` is the longest match possible for `<.+>`.
show_regex_match("Remember the numbers <311> and <911>", "<.+>")
```

```
Remember the numbers <311> and <911>
```

In many cases, using a more specific character class prevents these false matches:

```
show_regex_match("Remember the numbers <311> and <911>", "<[0-
9]+>")
```

```
Remember the numbers <311> and <911>
```

Anchoring¶

Sometimes a pattern should only match at the beginning or end of a string. The special character `^` anchors the regex to match only if the pattern appears at the beginning of the string; the special character `$` anchors the regex to match only if the pattern occurs at the end of the string. For example the regex `well$` only matches an appearance of `well` at the end of the string.

```
show_regex_match('well, well, well', r"well$")
```

```
well, well, well
```

Using both `^` and `$` requires the regex to match the full string.

```
phone_regex = r"^[0-9]{3}-[0-9]{3}-[0-9]{4}$"
show_regex_match('382-384-3840', phone_regex)
```

```
382-384-3840
```

```
# No match
show_regex_match('You can call me at 382-384-3840.',
phone_regex)
```

```
You can call me at 382-384-3840.
```

Escaping Meta Characters¶

All regex meta characters have special meaning in a regular expression. To match meta characters as literals, we escape them using the `\` character.

```
# `[` is a meta character and requires escaping
show_regex_match("Call me at [382-384-3840].", "\[")
```

```
Call me at [382-384-3840].
```

```
# `.` is a meta character and requires escaping
show_regex_match("Call me at [382-384-3840].", "\.")
```

```
Call me at [382-384-3840].
```

Reference Tables¶

We have now covered the most important pieces of regex syntax and meta characters. For a more complete reference, we include the tables below.

Meta Characters

This table includes most of the important *meta characters*, which help us specify certain patterns we want to match in a string.

<table>
<tr><th></th><th>Description</th><th>Example</th><th>Matches</th><th>Doesn't Match</th></tr>
<tr><td>.</td><td>Any character except \n</td><td>...</td><td>abc</td><td>ab
abcd</td></tr>
<tr><td>[ ]</td><td>Any character inside brackets</td><td>[cb.]ar</td><td>car
.ar</td><td>jar</td></tr>
<tr><td>[^]</td><td>Any character not inside brackets</td><td>[^b]ar</td><td>car
par</td><td>bar
ar</td></tr>
<tr><td>*</td><td>≥ 0 or more of last symbol</td><td>[pb]*ark</td><td>bbark
ark</td><td>dark</td></tr>
<tr><td>+</td><td>≥ 1 or more of last symbol</td><td>[pb]+ark</td><td>bbpark
bark</td><td>dark
ark</td></tr>
<tr><td>?</td><td>0 or 1 of last symbol</td><td>s?he</td><td>she
he</td><td>the</td></tr>
<tr><td>{n}</td><td>Exactly n of last symbol</td><td>hello{3}</td><td>hellooo</td><td>hello</td></tr>
<tr><td>|</td><td>Pattern before or after bar</td><td>we|[ui]s</td><td>we
us
is</td><td>e
s</td></tr>
<tr><td>\</td><td>Escapes next character</td><td><code>\[hi\]</code></td><td>[hi]</td><td>hi</td></tr>
<tr><td>^</td><td>Beginning of line</td><td>^ark</td><td>ark two</td><td>dark</td></tr>
<tr><td>\$</td><td>End of line</td><td>ark$</td><td>noahs ark</td><td>noahs arks</td></tr>
</table>

Shorthand Character Sets

Some commonly used character sets have shorthands.

Bracket Form	Shorthand	Description
[a-zA-Z0-9]	\w	Alphanumeric character
[^a-zA-Z0-9]	\W	Not an alphanumeric character
[0-9]	\d	Digit
[^0-9]	\D	Not a digit
[\t\n\f\r\p{Z}]	\s	Whitespace
[^\t\n\f\r\p{Z}]	\S	Not whitespace

Summary¶

Almost all programming languages have a library to match patterns using regular expressions, making them useful regardless of the specific language. In this section, we introduce regex syntax and the most useful meta characters.

Show Widgets Open on DataHub

Table of Contents

Regex and Python¶

In this section, we introduce regex usage in Python using the built-in `re` module. Since we only cover a few of the most commonly used methods, you will find it useful to consult the official documentation on the `re` module as well.

`re.search` ¶

`re.search(pattern, string)` searches for a match of the regex `pattern` anywhere in `string`. It returns a truthy match object if the pattern is found; it returns `None` if not.

```
phone_re = r"[0-9]{3}-[0-9]{3}-[0-9]{4}"
text  = "Call me at 382-384-3840."
match = re.search(phone_re, text)
match
```

```
<_sre.SRE_Match object; span=(11, 23), match='382-384-3840'>
```

Although the returned match object has a variety of useful properties, we most commonly use `re.search` to test whether a pattern appears in a string.

```
if re.search(phone_re, text):
    print("Found a match!")
```

```
Found a match!
```

```
if re.search(phone_re, 'Hello world'):
    print("No match; this won't print")
```

Another commonly used method, `re.match(pattern, string)`, behaves the same as `re.search` but only checks for a match at the start of `string` instead of a match anywhere in the string.

`re.findall`¶

We use `re.findall(pattern, string)` to extract substrings that match a regex. This method returns a list of all matches of `pattern` in `string`.

```
gmail_re = r'[a-zA-Z0-9]+@gmail\.com'
text = '''
From: email1@gmail.com
To: email2@yahoo.com and email3@gmail.com
'''
re.findall(gmail_re, text)
```

```
['email1@gmail.com', 'email3@gmail.com']
```

Regex Groups¶

Using **regex groups**, we specify subpatterns to extract from a regex by wrapping the subpattern in parentheses `( )`. When a regex contains regex groups, `re.findall` returns a list of tuples that contain the subpattern contents.

For example, the following familiar regex extracts phone numbers from a string:

```
phone_re = r"[0-9]{3}-[0-9]{3}-[0-9]{4}"
text  = "Sam's number is 382-384-3840 and Mary's is 123-456-
7890."
re.findall(phone_re, text)
```

```
['382-384-3840', '123-456-7890']
```

To split apart the individual three or four digit components of a phone number, we can wrap each digit group in parentheses.

```
# Same regex with parentheses around the digit groups
phone_re = r"([0-9]{3})-([0-9]{3})-([0-9]{4})"
text  = "Sam's number is 382-384-3840 and Mary's is 123-456-
7890."
re.findall(phone_re, text)
```

```
[('382', '384', '3840'), ('123', '456', '7890')]
```

As promised, `re.findall` returns a list of tuples containing the individual components of the matched phone numbers.

`re.sub` ¶

`re.sub(pattern, replacement, string)` replaces all occurrences of `pattern` with `replacement` in the provided `string`. This method behaves like the Python string method `str.sub` but uses a regex to match patterns.

In the code below, we alter the dates to have a common format by substituting the date separators with a dash.

```
messy_dates = '03/12/2018, 03.13.18, 03/14/2018, 03:15:2018'
regex = r'[/.:]'
re.sub(regex, '-', messy_dates)
```

```
'03-12-2018, 03-13-18, 03-14-2018, 03-15-2018'
```

`re.split` ¶

`re.split(pattern, string)` splits the input `string` each time the regex `pattern` appears. This method behaves like the Python string method `str.split` but uses a regex to make the split.

In the code below, we use `re.split` to split chapter names from their page numbers in a table of contents for a book.

```
toc = '''
PLAYING PILGRIMS============3
A MERRY CHRISTMAS===========13
THE LAURENCE BOY============31
BURDENS=====================55
BEING NEIGHBORLY============76
'''.strip()

# First, split into individual lines
lines = re.split('\n', toc)
lines
```

```
['PLAYING PILGRIMS============3',
 'A MERRY CHRISTMAS===========13',
 'THE LAURENCE BOY============31',
 'BURDENS=====================55',
 'BEING NEIGHBORLY============76']
```

```
# Then, split into chapter title and page number
split_re = r'=+' # Matches any sequence of = characters
[re.split(split_re, line) for line in lines]
```

```
[['PLAYING PILGRIMS', '3'],
 ['A MERRY CHRISTMAS', '13'],
 ['THE LAURENCE BOY', '31'],
 ['BURDENS', '55'],
 ['BEING NEIGHBORLY', '76']]
```

Regex and pandas¶

Recall that `pandas` Series objects have a `.str` property that supports string manipulation using Python string methods. Conveniently, the `.str` property also supports some functions from the `re` module. We demonstrate basic regex usage in `pandas`, leaving the

complete method list to the `pandas` documentation on string methods.

We've stored the text of the first five sentences of the novel *Little Women* in the DataFrame below. We can use the string methods that `pandas` provides to extract the spoken dialog in each sentence.

```
little
```

	sentences
0	"Christmas won't be Christmas without any pres...
1	"It's so dreadful to be poor!" sighed Meg, loo...
2	"I don't think it's fair for some girls to hav...
3	"We've got Father and Mother, and each other,"...
4	The four young faces on which the firelight sh...

Since spoken dialog lies within double quotation marks, we create a regex that captures a double quotation mark, a sequence of any characters except a double quotation mark, and the closing quotation mark.

```
quote_re = r'"[^"]+"'
little['sentences'].str.findall(quote_re)
```

```
0    ["Christmas won't be Christmas without any pre...
1                          ["It's so dreadful to be poor!"]
2    ["I don't think it's fair for some girls to ha...
3     ["We've got Father and Mother, and each other,"]
4    ["We haven't got Father, and shall not have hi...
Name: sentences, dtype: object
```

Since the `Series.str.findall` method returns a list of matches, `pandas` also provides `Series.str.extract` and `Series.str.extractall` method to extract matches into a Series or DataFrame. These methods require the regex to contain at least one regex group.

```
# Extract text within double quotes
quote_re = r'"([^"]+)"'
spoken = little['sentences'].str.extract(quote_re)
spoken
```

```
0    Christmas won't be Christmas without any prese...
1                         It's so dreadful to be poor!
2    I don't think it's fair for some girls to have...
3         We've got Father and Mother, and each other,
4    We haven't got Father, and shall not have him ...
Name: sentences, dtype: object
```

We can add this series as a column of the `little` DataFrame:

```
little['dialog'] = spoken
little
```

	sentences	dialog
0	"Christmas won't be Christmas without any pres...	Christmas won't be Christmas without any prese...
1	"It's so dreadful to be poor!" sighed Meg, loo...	It's so dreadful to be poor!
2	"I don't think it's fair for some girls to hav...	I don't think it's fair for some girls to have...
3	"We've got Father and Mother, and each other,"...	We've got Father and Mother, and each other,
4	The four young faces on which the firelight sh...	We haven't got Father, and shall not have him ...

We can confirm that our string manipulation behaves as expected for the last sentence in our DataFrame by printing the original and extracted text:

```
print(little.loc[4, 'sentences'])
```

```
The four young faces on which the firelight shone brightened at
the cheerful words, but darkened again as Jo said sadly, "We
haven't got Father, and shall not have him for a long time."
```

```
print(little.loc[4, 'dialog'])
```

```
We haven't got Father, and shall not have him for a long time.
```

Summary¶

The `re` module in Python provides a useful group of methods for manipulating text using regular expressions. When working with DataFrames, we often use the analogous string manipulation methods implemented in `pandas`.

For the complete documentation on the `re` module, see https://docs.python.org/3/library/re.html

For the complete documentation on `pandas` string methods, see https://pandas.pydata.org/pandas-docs/stable/text.html

Relational Databases and SQL

Thus far we have worked with datasets that are stored as text files on a computer. While useful for analysis of small datasets, using text files to store data presents challenges for many real-world use cases.

Many datasets are collected by multiple people—a team of data scientists, for example. If the data are stored in text files, however, the team will likely have to send and download new versions of the files each time the data are updated. Text files alone do not provide a consistent point of data retrieval for multiple analysts to use. This issue, among others, makes text files difficult to use for larger datasets or teams.

We often turn to relational database management systems (RDBMSs) to store data, such as MySQL or PostgreSQL. To work with these systems, we use a query language called SQL instead of Python. In this chapter, we discuss the relational database model and introduce SQL.

Show Widgets Open on DataHub

Table of Contents

The Relational Model¶

A **database** is an organized collection of data. In the past, data was stored in specialized data structures that were designed for specific tasks. For example, airlines might record flight bookings in a different format than a bank managing an account ledger. In 1969, Ted Codd introduced the relational model as a general method of storing data. Data is stored in two-dimensional tables called **relations**, consisting of individual observations in each row (commonly referred to as **tuples**). Each tuple is a structured data item that represents the relationship between certain **attributes** (columns). Each attribute of a relation has a name and data type.

Consider the `purchases` relation below:

purchases

name	product	retailer	date purchased
Samantha	iPod	Best Buy	June 3, 2016
Timothy	Chromebook	Amazon	July 8, 2016
Jason	Surface Pro	Target	October 2, 2016

In `purchases` , each tuple represents the relationship between the `name` , `product` , `retailer` , and `date purchased` attributes.

A relation's *schema* contains its column names, data types, and constraints. For example, the schema of the `purchases` table states that the columns are `name` , `product` , `retailer` , and `date purchased` ; it also states that each column contains text.

The following `prices` relation shows the price of certain gadgets at a few retail stores:

prices

retailer	product	price
Best Buy	Galaxy S9	719.00
Best Buy	iPod	200.00
Amazon	iPad	450.00
Amazon	Battery pack	24.87
Amazon	Chromebook	249.99
Target	iPod	215.00
Target	Surface Pro	799.00
Target	Google Pixel 2	659.00
Walmart	Chromebook	238.79

We can then reference both tables simultaneously to determine how much Samantha, Timothy, and Jason paid for their respective gadgets (assuming prices at each store stay constant over time). Together, the two tables form a **relational database**, which is a collection of one or more relations. The schema of the entire database is the set of schemas of the individual relations in the database.

Relational Database Management Systems¶

A relational database can be simply described as a set of tables containing rows of individual data entries. A relational database management system (RDBMSs) provides an interface to a relational database. Oracle, MySQL, and PostgreSQL are three of the most commonly used RDBMSs used in practice today.

Relational database management systems give users the ability to add, edit, and remove data from databases. These systems provide several key benefits over using a collection of text files to store data, including:

1. Reliable data storage: RDBMSs protect against data corruption from system failures or crashes.
2. Performance: RDBMSs often store data more efficiently than text files and have well-developed algorithms for querying data.
3. Data management: RDBMSs implement access control, preventing unauthorized users from accessing sensitive datasets.
4. Data consistency: RDBMSs can impose constraints on the data entered—for example, that a column `GPA` only contains floats between 0.0 and 4.0.

To work with data stored in a RDBMS, we use the SQL programming language.

RDBMS vs. pandas¶

How do RDBMSs and the `pandas` Python package differ? First, `pandas` is not concerned about data storage. Although DataFrames can read and write from multiple data formats, `pandas` does not dictate how the data are actually stored on the underlying computer like a RDBMS does. Second, `pandas` primarily provides methods for manipulating data while RDBMSs handle both data storage and data manipulation, making them more suitable for larger datasets. A typical rule of thumb is to use a RDBMS for datasets larger than several gigabytes. Finally, `pandas` requires knowledge of Python in order to use, whereas RDBMSs require knowledge of SQL. Since SQL is simpler to learn than Python, RDBMSs allow less technical users to store and query data, a handy trait.

Show Widgets Open on DataHub

Table of Contents

SQL¶

SQL (Structured Query Language) is a programming language that has operations to define, logically organize, manipulate, and perform calculations on data stored in a relational database management system (RDBMS).

SQL is a declarative language. This means that the user only needs to specify *what* kind of data they want, not *how* to obtain it. An example is shown below, with an imperative example for comparison:

- **Declarative**: Compute the table with columns "x" and "y" from table "A" where the values in "y" are greater than 100.00.
- **Imperative**: For each record in table "A", check if the record contains a value of "y" greater than 100. If so, then store the record's "x" and "y" attributes in a new table. Return the new table.

In this chapter, we will write SQL queries as Python strings, then use `pandas` to execute the SQL query and read the result into a `pandas` DataFrame. As we walk through the basics of SQL syntax, we'll also occasionally show `pandas` equivalents for comparison purposes.

Executing SQL Queries through `pandas` ¶

To execute SQL queries from Python, we will connect to a database using the sqlalchemy library. Then we can use the `pandas` function pd.read_sql to execute SQL queries through this connection.

```
import sqlalchemy

# pd.read_sql takes in a parameter for a SQLite engine, which we
create below
sqlite_uri = "sqlite:///sql_basics.db"
sqlite_engine = sqlalchemy.create_engine(sqlite_uri)
```

This database contains one relation: `prices`. To display the relation we run a SQL query. Calling `read_sql` will execute the SQL query on the RDBMS, then return the results in a `pandas` DataFrame.

```
sql_expr = """
SELECT *
FROM prices
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	product	price
0	Best Buy	Galaxy S9	719.00
1	Best Buy	iPod	200.00
2	Amazon	iPad	450.00
3	Amazon	Battery pack	24.87
4	Amazon	Chromebook	249.99
5	Target	iPod	215.00
6	Target	Surface Pro	799.00
7	Target	Google Pixel 2	659.00
8	Walmart	Chromebook	238.79

Later in this section we will compare SQL queries with `pandas` method calls so we've created an identical DataFrame in `pandas`.

```
prices
```

	retailer	product	price
0	Best Buy	Galaxy S9	719.00
1	Best Buy	iPod	200.00
2	Amazon	iPad	450.00
3	Amazon	Battery pack	24.87
4	Amazon	Chromebook	249.99
5	Target	iPod	215.00
6	Target	Surface Pro	799.00
7	Target	Google Pixel 2	659.00
8	Walmart	Chromebook	238.79

SQL Syntax¶

All SQL queries take the general form below:

```
SELECT [DISTINCT] <column expression list>
FROM <relation>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number>]
```

Note that:

1. **Everything in [square brackets] is optional.** A valid SQL query only needs a `SELECT` and a `FROM` statement.
2. **SQL SYNTAX IS GENERALLY WRITTEN IN CAPITAL LETTERS.** Although capitalization isn't required, it is common practice to write SQL syntax in capital letters. It also helps to visually structure your query for others to read.
3. `FROM` query blocks can reference one or more tables, although in this section we will only look at one table at a time for simplicity.

SELECT and FROM¶

The two mandatory statements in a SQL query are:

- `SELECT` indicates the columns that we want to view.
- `FROM` indicates the tables from which we are selecting these columns.

To display the entire `prices` table, we run:

```
sql_expr = """
SELECT *
FROM prices
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	product	price
0	Best Buy	Galaxy S9	719.00
1	Best Buy	iPod	200.00
2	Amazon	iPad	450.00
3	Amazon	Battery pack	24.87
4	Amazon	Chromebook	249.99
5	Target	iPod	215.00
6	Target	Surface Pro	799.00
7	Target	Google Pixel 2	659.00
8	Walmart	Chromebook	238.79

`SELECT *` returns every column in the original relation. To display only the retailers that are represented in `prices`, we add the `retailer` column to the `SELECT` statement.

```
sql_expr = """
SELECT retailer
FROM prices
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer
0	Best Buy
1	Best Buy
2	Amazon
3	Amazon
4	Amazon
5	Target
6	Target
7	Target
8	Walmart

If we want a list of unique retailers, we can call the `DISTINCT` function to omit repeated values.

```
sql_expr = """
SELECT DISTINCT(retailer)
FROM prices
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer
0	Best Buy
1	Amazon
2	Target
3	Walmart

This would be the functional equivalent of the following `pandas` code:

```
prices['retailer'].unique()
```

```
array(['Best Buy', 'Amazon', 'Target', 'Walmart'], dtype=object)
```

Each RDBMS comes with its own set of functions that can be applied to attributes in the `SELECT` list, such as comparison operators, mathematical functions and operators, and string functions and operators. In Data 100 we use PostgreSQL, a mature RDBMS that

comes with hundreds of such functions. The complete list is available here. Keep in mind that each RDBMS has a different set of functions for use in `SELECT`.

The following code converts all retailer names to uppercase and halves the product prices.

```
sql_expr = """
SELECT
    UPPER(retailer) AS retailer_caps,
    product,
    price / 2 AS half_price
FROM prices
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer_caps	product	half_price
0	BEST BUY	Galaxy S9	359.500
1	BEST BUY	iPod	100.000
2	AMAZON	iPad	225.000
3	AMAZON	Battery pack	12.435
4	AMAZON	Chromebook	124.995
5	TARGET	iPod	107.500
6	TARGET	Surface Pro	399.500
7	TARGET	Google Pixel 2	329.500
8	WALMART	Chromebook	119.395

Notice that we can **alias** the columns (assign another name) with `AS` so that the columns appear with this new name in the output table. This does not modify the names of the columns in the source relation.

WHERE¶

The `WHERE` clause allows us to specify certain constraints for the returned data; these constraints are often referred to as **predicates**. For example, to retrieve only gadgets that are under $500:

```
sql_expr = """
SELECT *
FROM prices
WHERE price < 500
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	product	price
0	Best Buy	iPod	200.00
1	Amazon	iPad	450.00
2	Amazon	Battery pack	24.87
3	Amazon	Chromebook	249.99
4	Target	iPod	215.00
5	Walmart	Chromebook	238.79

We can also use the operators `AND`, `OR`, and `NOT` to further constrain our SQL query. To find an item on Amazon without a battery pack under $300, we write:

```
sql_expr = """
SELECT *
FROM prices
WHERE retailer = 'Amazon'
    AND NOT product = 'Battery pack'
    AND price < 300
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	product	price
0	Amazon	Chromebook	249.99

The equivalent operation in `pandas` is:

```
prices[(prices['retailer'] == 'Amazon')
   & ~(prices['product'] == 'Battery pack')
   & (prices['price'] <= 300)]
```

	retailer	product	price
4	Amazon	Chromebook	249.99

There's a subtle difference that's worth noting: the index of the Chromebook in the SQL query is 0, whereas the corresponding index in the DataFrame is 4. This is because SQL queries always return a new table with indices counting up from 0, whereas `pandas` subsets a portion of the DataFrame `prices` and returns it with the original indices. We can use pd.DataFrame.reset_index to reset the indices in `pandas`.

Aggregate Functions¶

So far, we've only worked with data from the existing rows in the table; that is, all of our returned tables have been some subset of the entries found in the table. But to conduct data analysis, we'll want to compute aggregate values over our data. In SQL, these are called **aggregate functions**.

If we want to find the average price of all gadgets in the `prices` relation:

```
sql_expr = """
SELECT AVG(price) AS avg_price
FROM prices
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	avg_price
0	395.072222

Equivalently, in `pandas`:

```
prices['price'].mean()
```

```
395.0722222222222
```

A complete list of PostgreSQL aggregate functions can be found here. Though we're using PostgreSQL as our primary version of SQL in Data 100, keep in mind that there are many other variations of SQL (MySQL, SQLite, etc.) that use different function names and have different functions available.

GROUP BY and HAVING¶

With aggregate functions, we can execute more complicated SQL queries. To operate on more granular aggregate data, we can use the following two clauses:

- `GROUP BY` takes a list of columns and groups the table like the pd.DataFrame.groupby function in `pandas`.
- `HAVING` is functionally similar to `WHERE`, but is used exclusively to apply predicates to aggregated data. (Note that in order to use `HAVING`, it must be preceded by a `GROUP BY` clause.)

Important: When using `GROUP BY`, all columns in the `SELECT` clause must be either listed in the `GROUP BY` clause or have an aggregate function applied to them.

We can use these statements to find the maximum price at each retailer.

```
sql_expr = """
SELECT retailer, MAX(price) as max_price
FROM prices
GROUP BY retailer
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	max_price
0	Amazon	450.00
1	Best Buy	719.00
2	Target	799.00
3	Walmart	238.79

Let's say we have a client with expensive taste and only want to find retailers that sell gadgets over $700. Note that we must use `HAVING` to define predicates on aggregated columns; we can't use `WHERE` to filter an aggregated column. To compute a list of retailers and accompanying prices that satisfy our needs, we run:

```
sql_expr = """
SELECT retailer, MAX(price) as max_price
FROM prices
GROUP BY retailer
HAVING max_price > 700
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	max_price
0	Best Buy	719.0
1	Target	799.0

For comparison, we recreate the same table in `pandas`:

```
max_prices = prices.groupby('retailer').max()
max_prices.loc[max_prices['price'] > 700, ['price']]
```

	price
retailer	
Best Buy	719.0
Target	799.0

ORDER BY and LIMIT¶

These clauses allow us to control the presentation of the data:

- `ORDER BY` lets us present the data in lexicographic order of column values. By default, ORDER BY uses ascending order (`ASC`) but we can specify descending order using `DESC`.
- `LIMIT` controls how many tuples are displayed.

Let's display the three cheapest items in our `prices` table:

```
sql_expr = """
SELECT *
FROM prices
ORDER BY price ASC
LIMIT 3
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	retailer	product	price
0	Amazon	Battery pack	24.87
1	Best Buy	iPod	200.00
2	Target	iPod	215.00

Note that we didn't have to include the `ASC` keyword since `ORDER BY` returns data in ascending order by default. For comparison, in `pandas` :

```
prices.sort_values('price').head(3)
```

	retailer	product	price
3	Amazon	Battery pack	24.87
1	Best Buy	iPod	200.00
5	Target	iPod	215.00

(Again, we see that the indices are out of order in the `pandas` DataFrame. As before, `pandas` returns a view on our DataFrame `prices` , whereas SQL is displaying a new table each time that we execute a query.)

Conceptual SQL Evaluation¶

Clauses in a SQL query are executed in a specific order. Unfortunately, this order differs from the order that the clauses are written in a SQL query. From first executed to last:

1. `FROM` : One or more source tables
2. `WHERE` : Apply selection qualifications (eliminate rows)
3. `GROUP BY` : Form groups and aggregate
4. `HAVING` : Eliminate groups
5. `SELECT` : Select columns

Note on `WHERE` vs. `HAVING` : Since the `WHERE` clause is processed before applying `GROUP BY` , the `WHERE` clause cannot make use of aggregated values. To define predicates based on aggregated values, we must use the `HAVING` clause.

Summary¶

We have introduced SQL syntax and the most important SQL statements needed to conduct data analysis using a relational database management system.

Show Widgets Open on DataHub

Table of Contents

SQL Joins¶

In `pandas` we use the `pd.merge` method to join two tables using matching values in their columns. For example:

```
pd.merge(table1, table2, on='commmon_column')
```

In this section, we introduce SQL joins. SQL joins are used to combine multiple tables in a relational database.

Suppose we are cat store owners with a database for the cats we have in our store. We have **two** different tables: `names` and `colors`. The `names` table contains the columns `cat_id`, a unique number assigned to each cat, and `name`, the name for the cat. The `colors` table contains the columns `cat_id` and `color`, the color of each cat.

Note that there are some missing rows from both tables - a row with `cat_id` 3 is missing from the `names` table, and a row with `cat_id` 4 is missing from the `colors` table.

names

cat_id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

colors

cat_id	color
0	orange
1	black
2	calico
3	white

To compute the color of the cat named Apricot, we have to use information in both tables. We can *join* the tables on the `cat_id` column, creating a new table with both `name` and `color`.

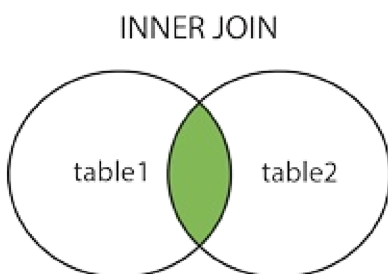
Joins¶

A join combines tables by matching values in their columns.

There are four main types of joins: inner joins, outer joins, left joins, and right joins. Although all four combine tables, each one treats non-matching values differently.

Inner Join¶

Definition: In an inner join, the final table only contains rows that have matching columns in **both** tables.



Example: We would like to join the `names` and `colors` tables together to match each cat with its color. Since both tables contain a `cat_id` column that is the unique identifier for a cat, we can use an inner join on the `cat_id` column.

SQL: To write an inner join in SQL we modify our `FROM` clause to use the following syntax:

```
SELECT ...
FROM <TABLE_1>
    INNER JOIN <TABLE_2>
    ON <...>
```

For example:

```
SELECT *
FROM names AS N
    INNER JOIN colors AS C
    ON N.cat_id = C.cat_id;
```

	cat_id	name	cat_id	color
0	0	Apricot	0	orange
1	1	Boots	1	black
2	2	Cally	2	calico

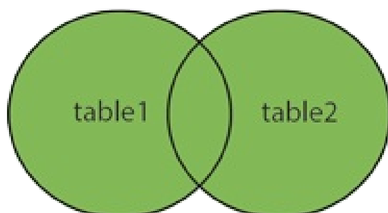
You may verify that each cat name is matched with its color. Notice that the cats with `cat_id` 3 and 4 are not present in our resulting table because the `colors` table doesn't have a row with `cat_id` 4 and the `names` table doesn't have a row with `cat_id` 3. In an inner join, if a row doesn't have a matching value in the other table, the row is not included in the final result.

Assuming we have a DataFrame called `names` and a DataFrame called `colors`, we can conduct an inner join in `pandas` by writing:

```
pd.merge(names, colors, how='inner', on='cat_id')
```

Full/Outer Join¶

Definition: In a full join (sometimes called an outer join), **all records from both tables** are included in the joined table. If a row doesn't have a match in the other table, the missing values are filled in with `NULL`.



Example: As before, we join the `names` and `colors` tables together to match each cat with its color. This time, we want to keep all rows in either table even if there isn't a match.

SQL: To write an outer join in SQL we modify our `FROM` clause to use the following syntax:

```
SELECT ...
FROM <TABLE_1>
    FULL JOIN <TABLE_2>
    ON <...>
```

For example:

```
SELECT name, color
FROM names N
    FULL JOIN colors C
    ON N.cat_id = C.cat_id;
```

cat_id	name	color
0	Apricot	orange
1	Boots	black
2	Cally	calico
3	NULL	white
4	Eugene	NULL

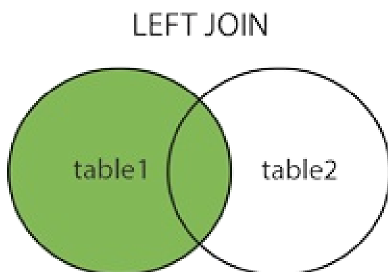
Notice that the final output contains the entries with `cat_id` 3 and 4. If a row does not have a match, it is still included in the final output and any missing values are filled in with `NULL`.

In `pandas`:

```
pd.merge(names, colors, how='outer', on='cat_id')
```

Left Join¶

Definition: In a left join, all records from the **left table** are included in the joined table. If a row doesn't have a match in the right table, the missing values are filled in with `NULL`.



Example: As before, we join the `names` and `colors` tables together to match each cat with its color. This time, we want to keep all the cat names even if a cat doesn't have a matching color.

SQL: To write an left join in SQL we modify our `FROM` clause to use the following syntax:

```
SELECT ...
FROM <TABLE_1>
    LEFT JOIN <TABLE_2>
    ON <...>
```

For example:

```
SELECT name, color
FROM names N
    LEFT JOIN colors C
    ON N.cat_id = C.cat_id;
```

cat_id	name	color
0	Apricot	orange
1	Boots	black
2	Cally	calico
4	Eugene	NULL

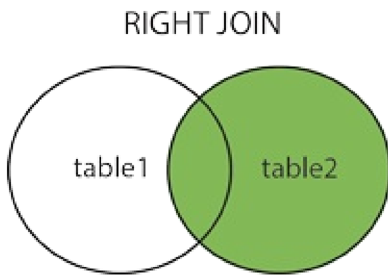
Notice that the final output includes all four cat names. Three of the `cat_id` s in the `names` relation had matching `cat_id` s in the `colors` table and one did not (Eugene). The cat name that did not have a matching color has `NULL` as its color.

In `pandas` :

```
pd.merge(names, colors, how='left', on='cat_id')
```

Right Join¶

Definition: In a right join, all records from the **right table** are included in the joined table. If a row doesn't have a match in the left table, the missing values are filled in with `NULL` .



Example: As before, we join the `names` and `colors` tables together to match each cat with its color. This time, we want to keep all the cat color even if a cat doesn't have a matching name.

SQL: To write a right join in SQL we modify our `FROM` clause to use the following syntax:

```
SELECT ...
FROM <TABLE_1>
    RIGHT JOIN <TABLE_2>
    ON <...>
```

For example:

```
SELECT name, color
FROM names N
    RIGHT JOIN colors C
    ON N.cat_id = C.cat_id;
```

cat_id	name	color
0	Apricot	orange
1	Boots	black
2	Cally	calico
3	NULL	white

This time, observe that the final output includes all four cat colors. Three of the `cat_id` s in the `colors` relation had matching `cat_id` s in the `names` table and one did not (white). The cat color that did not have a matching name has `NULL` as its name.

You may also notice that a right join produces the same result a left join with the table order swapped. That is, `names` left joined with `colors` is the same as `colors` right joined with `names` . Because of this, some SQL engines (such as SQLite) do not support right joins.

In `pandas` :

```
pd.merge(names, colors, how='right', on='cat_id')
```

Implicit Inner Joins¶

There are typically multiple ways to accomplish the same task in SQL just as there are multiple ways to accomplish the same task in Python. We point out one other method for writing an inner join that appears in practice called an *implicit join*. Recall that we previously wrote the following to conduct an inner join:

```
SELECT *
FROM names AS N
    INNER JOIN colors AS C
    ON N.cat_id = C.cat_id;
```

An implicit inner join has a slightly different syntax. Notice in particular that the `FROM` clause uses a comma to select from two tables and that the query includes a `WHERE` clause to specify the join condition.

```
SELECT *
FROM names AS N, colors AS C
WHERE N.cat_id = C.cat_id;
```

When multiple tables are specified in the `FROM` clause, SQL creates a table containing every combination of rows from each table. For example:

```
sql_expr = """
SELECT *
FROM names N, colors C
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	cat_id	name	cat_id	color
0	0	Apricot	0	orange
1	0	Apricot	1	black
2	0	Apricot	2	calico
3	0	Apricot	3	white
4	1	Boots	0	orange
5	1	Boots	1	black
6	1	Boots	2	calico
7	1	Boots	3	white
8	2	Cally	0	orange
9	2	Cally	1	black
10	2	Cally	2	calico
11	2	Cally	3	white
12	4	Eugene	0	orange
13	4	Eugene	1	black
14	4	Eugene	2	calico
15	4	Eugene	3	white

This operation is often called a *Cartesian product*: each row in the first table is paired with every row in the second table. Notice that many rows contain cat colors that are not matched properly with their names. The additional `WHERE` clause in the implicit join filters out rows that do not have matching `cat_id` values.

```
SELECT *
FROM names AS N, colors AS C
WHERE N.cat_id = C.cat_id;
```

	cat_id	name	cat_id	color
0	0	Apricot	0	orange
1	1	Boots	1	black
2	2	Cally	2	calico

Joining Multiple Tables¶

To join multiple tables, extend the `FROM` clause with additional `JOIN` operators. For example, the following table `ages` includes data about each cat's age.

cat_id	age
0	4
1	3
2	9
4	20

To conduct an inner join on the `names`, `colors`, and `ages` table, we write:

```
# Joining three tables

sql_expr = """
SELECT name, color, age
    FROM names n
    INNER JOIN colors c ON n.cat_id = c.cat_id
    INNER JOIN ages a ON n.cat_id = a.cat_id;
"""
pd.read_sql(sql_expr, sqlite_engine)
```

	name	color	age
0	Apricot	orange	4
1	Boots	black	3
2	Cally	calico	9

Summary¶

We have covered the four main types of SQL joins: inner, full, left, and right joins. We use all four joins to combine information in separate relations, and each join differs only in how it handles non-matching rows in the input tables.

Modeling and Estimation

> Essentially, all models are wrong, but some are useful.
>
> — George Box, Statistician (1919-2013)

We have covered question formulation, data cleaning, and exploratory data analysis, the first three steps of the data science lifecycle. We have also seen that EDA often reveals relationships between variables in our dataset. How do we decide whether a relationship is real or spurious? How do we use these relationships to make reliable predictions about the future? To answer these questions we will need the mathematical tools for modeling and estimation.

A model is an **idealized** representation of a system. For example, if we drop a ball off the Leaning Tower of Pisa, we expect the ball to drop to the ground because we have a model of gravity. Our model of gravity also allows us to predict how long it will take the ball to hit the ground using the laws of projectile motion.

This model represents our system but is simplistic—for example, it leaves out the effects of air resistance, the gravitational effects of other celestial bodies, and the buoyancy of air. Because of these unconsidered factors, our model will almost always make incorrect predictions in real life! Still, the simple model of gravity is accurate enough in so many situations that it's widely used and taught today.

In data science, we often use data to create models to estimate how the world will behave in the future. How do we choose a model? How do we decide whether we need a more complicated model? We will explore these questions in this chapter.

Show Widgets Open on DataHub

Table of Contents

Models¶

In the United States, many diners will leave a tip for their waiter or waitress as the diners pay for the meal. Although it is customary to offer 15% of the total bill as tip, perhaps some restaurants have more generous patrons than others.

One particular waiter was so interested in how much tip he could expect to get that he collected a simple random sample of information about the tables he served.

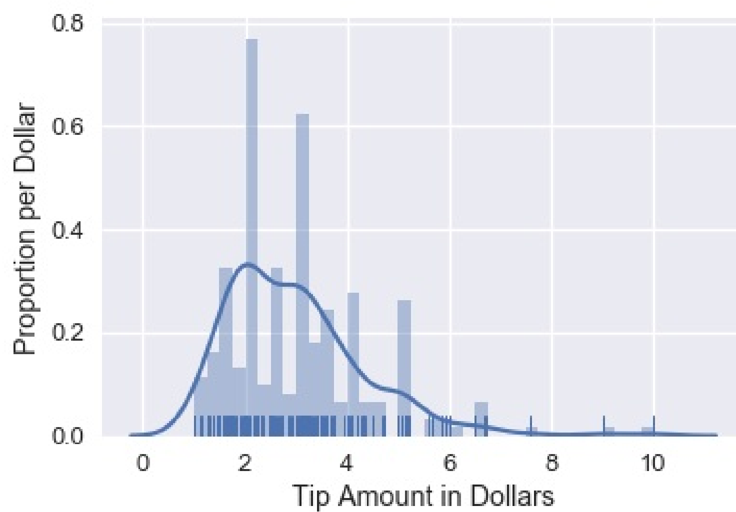
```
tips = sns.load_dataset('tips')
tips
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
...	...	...	...	...	...	...	...
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

We can plot a histogram of the tip amounts:

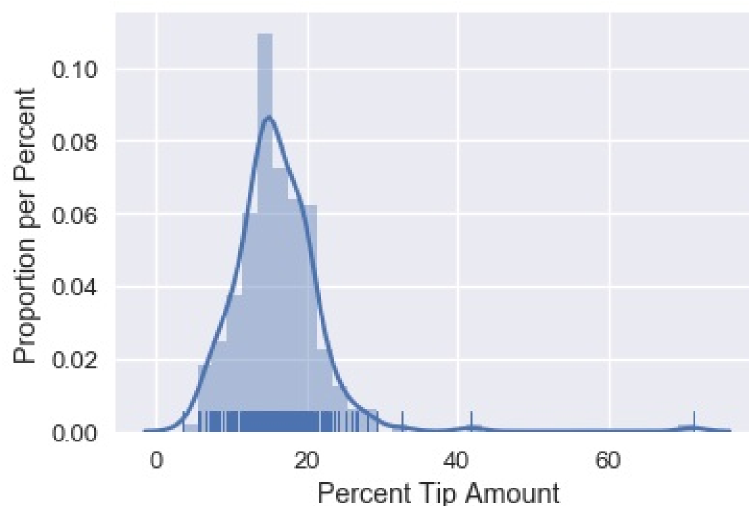
```
sns.distplot(tips['tip'], bins=np.arange(0, 10.1, 0.25),
rug=True)
plt.xlabel('Tip Amount in Dollars')
plt.ylabel('Proportion per Dollar');
```



There are already some interesting patterns in the data. For example, there is a clear mode at \$2 and most tips seem to be in multiples of \$0.50.

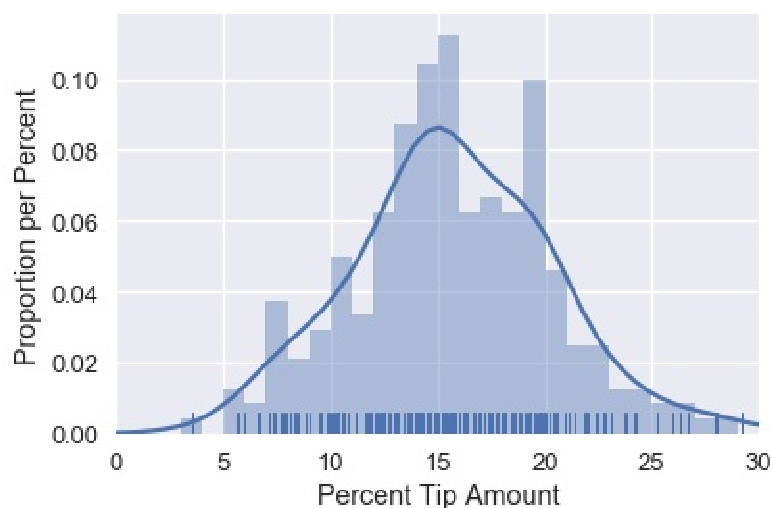
For now, we are most interested in the percent tip: the tip amount divided by the bill amount. We can create a column in our DataFrame for this variable and show its distribution.

```
tips['pcttip'] = tips['tip'] / tips['total_bill'] * 100
sns.distplot(tips['pcttip'], rug=True)
plt.xlabel('Percent Tip Amount')
plt.ylabel('Proportion per Percent');
```



It looks like one table left our waiter a tip of 70%! However, most of the tips percentages are under 30%. Let's zoom into that part of the distribution.

```
sns.distplot(tips['pcttip'], bins=np.arange(30), rug=True)
plt.xlim(0, 30)
plt.xlabel('Percent Tip Amount')
plt.ylabel('Proportion per Percent');
```



We can the distribution is roughly centered at 15% with another potential mode at 20%. Suppose our waiter is interested in predicting how much percent tip he will get from a given table. To address this question, we can create a model for how much tip the waiter will get.

A Simple Model¶

The simplest model possible is to completely ignore the data and state that since the convention in the U.S. is to give 15% tip, the waiter will always get 15% tip from his tables. While simple, let's use this model to define some variables that we'll use later on.

This model assumes that there is one true percentage tip that all tables, past and future, will give the waiter. This is the *population parameter* for the percent tip, which we will denote by θ^*.

After making this assumption, our model then says that our guess for θ^* is 15%. We will use θ to represent our estimate.

In mathematical notation, our model states that:

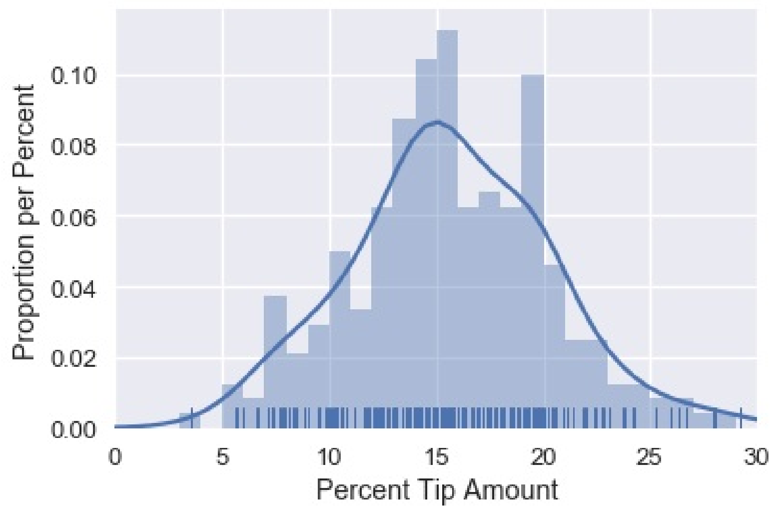
$$\theta = 15$$

This model is obviously wrong—if the model were accurate, every table in our dataset should have given the waiter exactly 15% tip. However, this model will probably make a reasonable guess for most scenarios. In fact, if we had no data on previous tables and their percent tips this model would be a good choice. Still, the question remains: what if 10% is a better estimate? Or 20%? Without data, we are mostly limited to blind guesses.

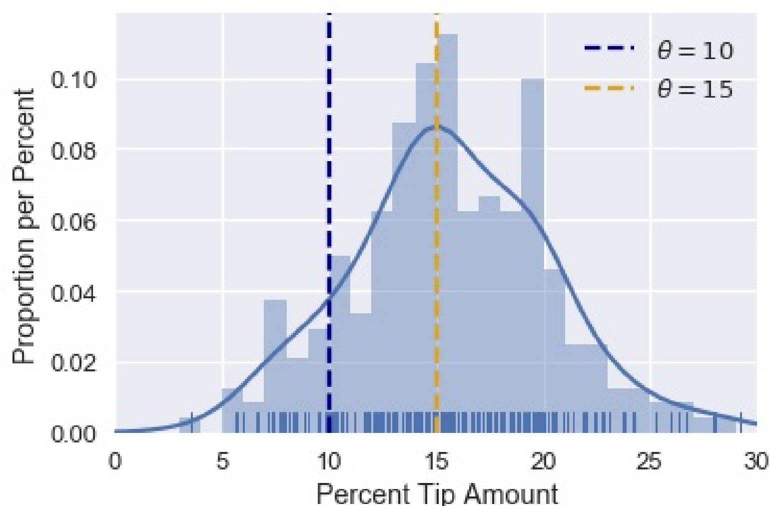
However, since our waiter's dataset was sampled at random from our population of interest, our sample will likely resemble the population. This key insight allows us to use our sample to evaluate the accuracy of our model. In particular, we can use the sample of tip percentages to decide whether 10%, 15%, 20%, or some other percent is the best choice for our estimate θ.

Cost Function Intuition¶

The distribution of tip percents from our dataset is replicated below for convenience.



Let's suppose we are trying to compare two choices for θ: 10% and 15%. We can mark both of these choices on our distribution:



Intuitively, it looks like choosing $ \theta = 15 $ makes more sense than $ \theta = 10 $ given our dataset. Why is this? When we look at the points in our data, we can see that more points fall close to 15 than they do to 10.

Although in this case it is clear that $ \theta = 15 $ is a better choice than $ \theta = 10 $, it is not so clear whether $ \theta = 15 $ is a better choice than $ \theta = 16 $. By assigning a specific number that measures how "good" a given choice of $ \theta $ is, we make precise choices between different values of $ \theta $. This is a mathematical function that takes in a value of $ \theta $ and the points in our dataset, outputting a single number that we will use to select the best value of $ \theta $ that we can.

We call this function a *cost function*.

Show Widgets Open on DataHub

Table of Contents

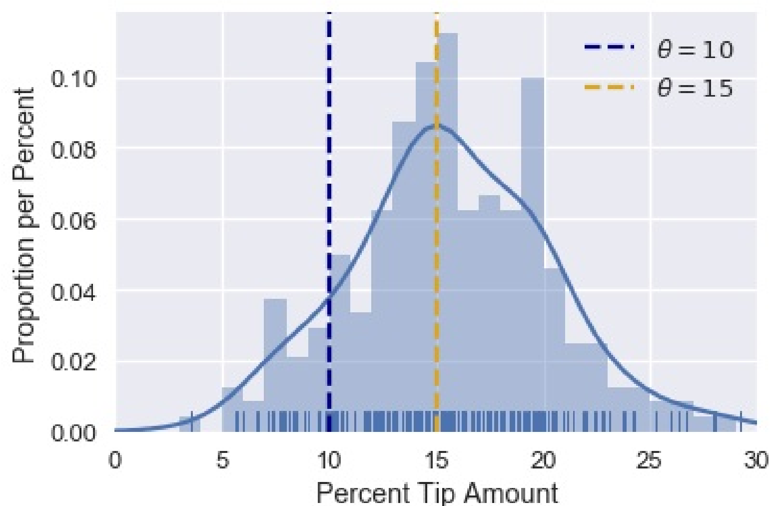
Cost Functions¶

For now, our model assumes that there is a single population tip percentage $ \theta^* $. We are trying to estimate this parameter, and we use the variable $ \theta $ to denote our estimate. Since our sample of tips is a random sample drawn from the population, we believe that using our sample to create an estimate $ \theta $ will give a value close to $ \theta^* $.

To precisely decide which value of $ \theta $ is best, we need to define a *cost function*. A cost function is a mathematical function that takes in an estimate $ \theta $ and the points in our dataset $y_1, y_2, \ldots, y_n$. It outputs a single number that we can use to choose between two different values of $ \theta $. In mathematical notation, we want to create the function:

$$ L(\theta, y_1, y_2, \ldots, y_n) =\ \ldots $$

By convention, the cost function outputs lower values for preferable values of $ \theta $ and larger values for worse values of $ \theta $. In the previous section, we compared $ \theta = 10 $ and $ \theta = 15 $.



Since $\theta = 15$ falls closer to most of the points, our cost function should output a small value for $\theta = 15$ and a larger value for $\theta = 10$.

Let's use this intuition to create a cost function.

Our First Cost Function: Mean Squared Error¶

We would like our choice of θ to fall close to the points in our dataset. Thus, we can define a cost function that outputs a larger value as θ gets further away from the points in the dataset. We start with a simple cost function called the *mean squared error*. Here's the idea:

1. We select a value of θ.
2. For each value in our dataset, take the squared difference between the value and theta: $(y_i - \theta)^2$. Squaring the difference in a simple way to convert negative differences into positive ones. We want to do this because if our point $y_i = 14$, $\theta = 10$ and $\theta = 18$ are equally far away from the point and are thus equally "bad".
3. To compute the final cost, take the average of each of the individual squared differences.

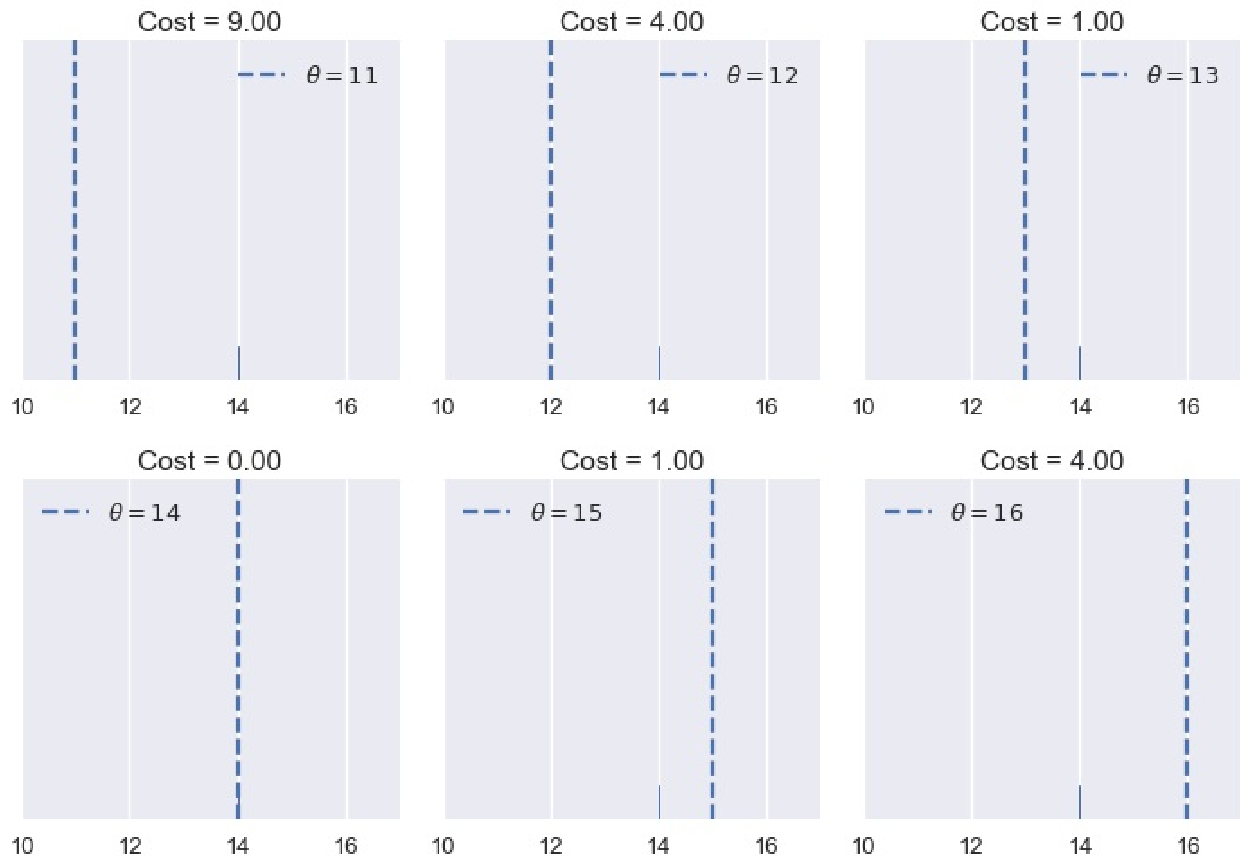
This gives us a final cost function of:

$$ \begin{aligned} L(\theta, y_1, y_2, \ldots, y_n) &= \text{average}\left\{ (y_1 - \theta)^2, (y_2 - \theta)^2, \ldots, (y_n - \theta)^2 \right\} \\ &= \frac{1}{n} \left((y_1 - \theta)^2 + (y_2 - \theta)^2 + \ldots + (y_n - \theta)^2 \right) \\ &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \end{aligned} $$

Creating a Python function to compute the loss is simple to do:

```
def mse_cost(theta, y_vals):
    return np.mean((y_vals - theta) ** 2)
```

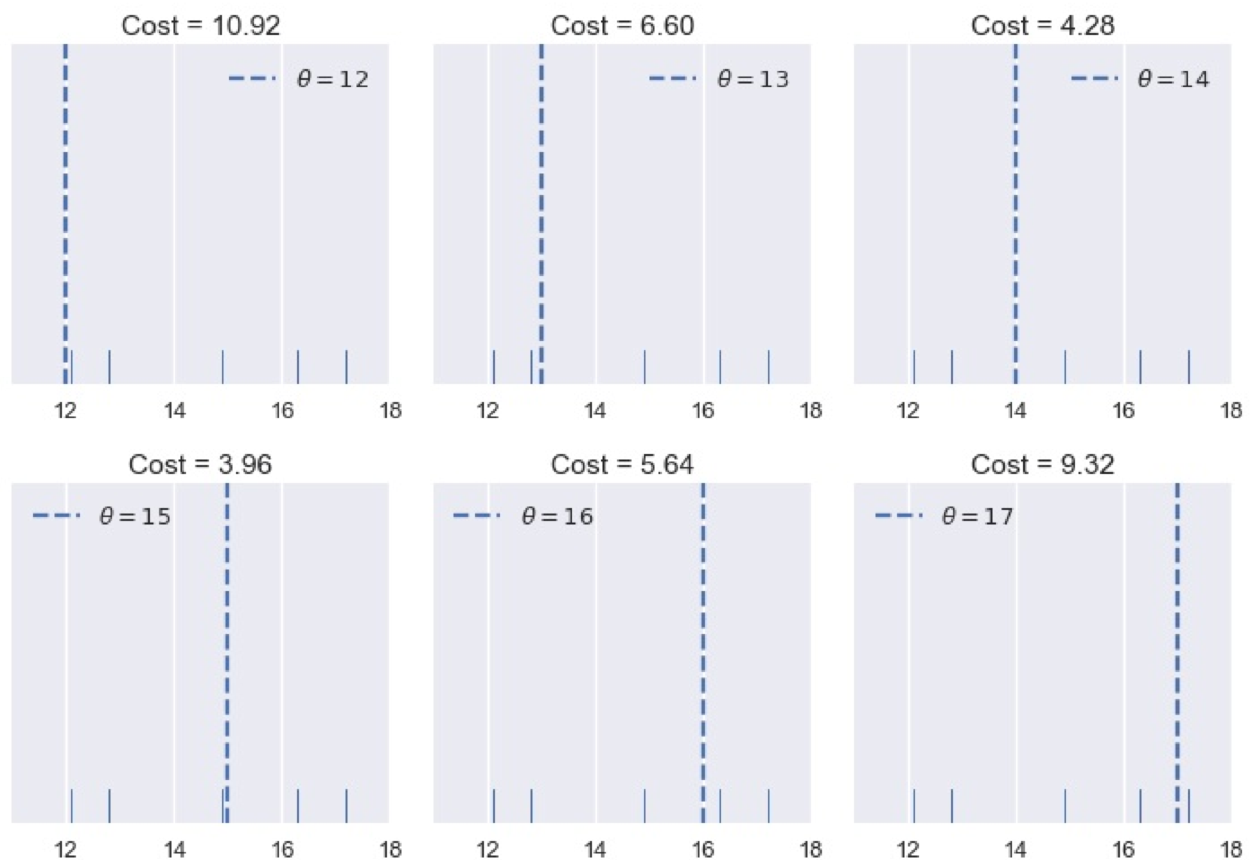
Let's see how this cost function behaves. Suppose we have a dataset only containing one point, $ y_1 = 14 $. We can try different values of $ \theta $ and see what the cost function outputs for each value.



You can also interactively try different values of $ \theta $ below. You should understand why the cost for $ \theta = 11 $ is many times higher than the cost for $ \theta = 13 $.

Show Widget

As we hoped, our cost is larger as $ \theta $ is further away from our data and is smallest when $ \theta $ falls exactly onto our data point. Let's now see how our mean squared error behaves when we have five points instead of one. Our data this time are: $ \{ 12.1, 12.8, 14.9, 16.3, 17.2 \} $.

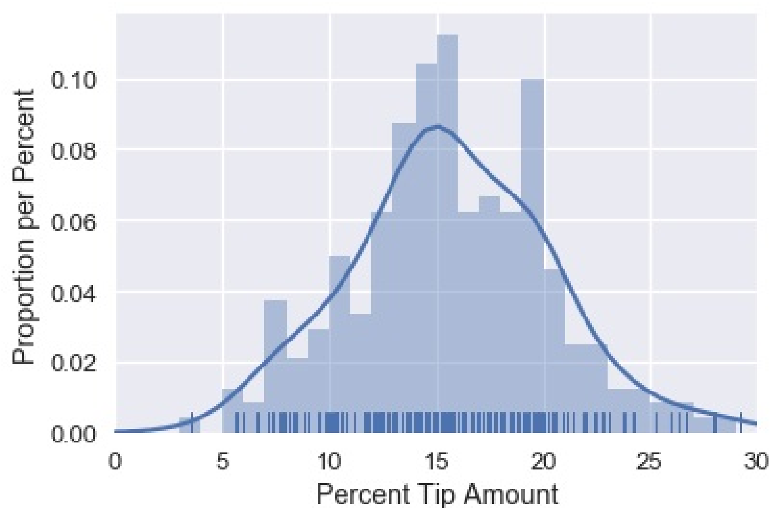


Of the values of $ \theta $ we tried $ \theta = 15 $ has the lowest cost. However, a value of $ \theta $ in between 14 and 15 might have an even lower cost than $ \theta = 15 $. See if you can find a better value of $ \theta $ using the interactive plot below.

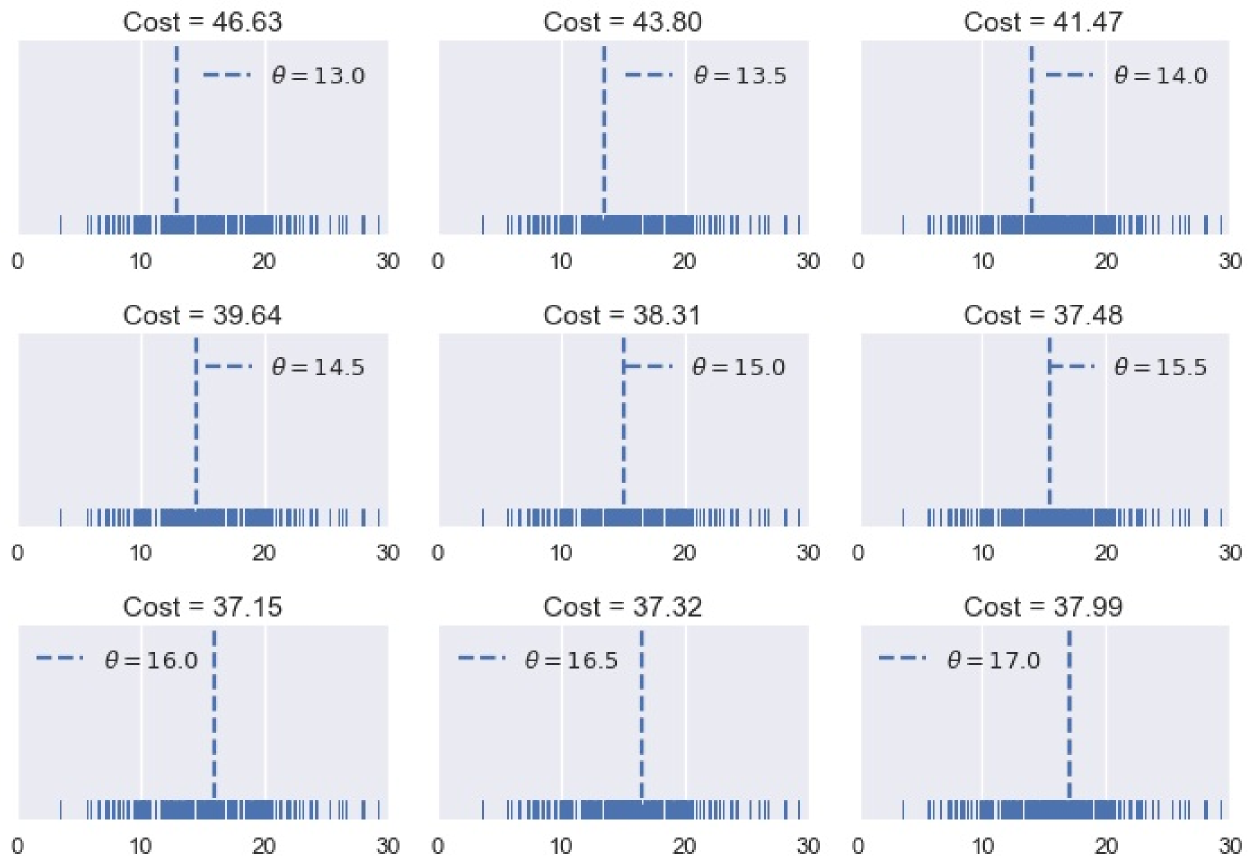
(How might we know for certain whether we've found the best value of $ \theta $? We will tackle this issue soon.)

Show Widget

The mean squared error cost function seems to be doing its job by penalizing values of $ \theta $ that are far away from the center of the data. Let's now see what the cost function outputs on the original dataset of tip percents. For reference, the original distribution of tip percents is plotted below:



Let's try some values of $ \theta $.



As before, we've created an interactive widget to test different values of $ \theta $.

Show Widget

It looks like the best value of $ \theta $ that we've tried so far is 16.00. This is slightly above our original blind guess of 15% tip. It appears that our waiter gets a bit more tip than we originally thought.

A Shorthand¶

We have defined our first cost function, the mean squared error cost (MSE). It computes high cost for values of $ \theta $ that are further away from the center of the data. Mathematically, this cost function is defined as:

$$ \begin{aligned} L(\theta, y_1, y_2, \ldots, y_n) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \end{aligned} $$

The cost function will compute different costs whenever we change either $ \theta $ or $ y_1, y_2, \ldots, y_n $. We've seen this happen when we tried different values of $ \theta $ and when we added new data points (changing $ y_1, y_2, \ldots, y_n $).

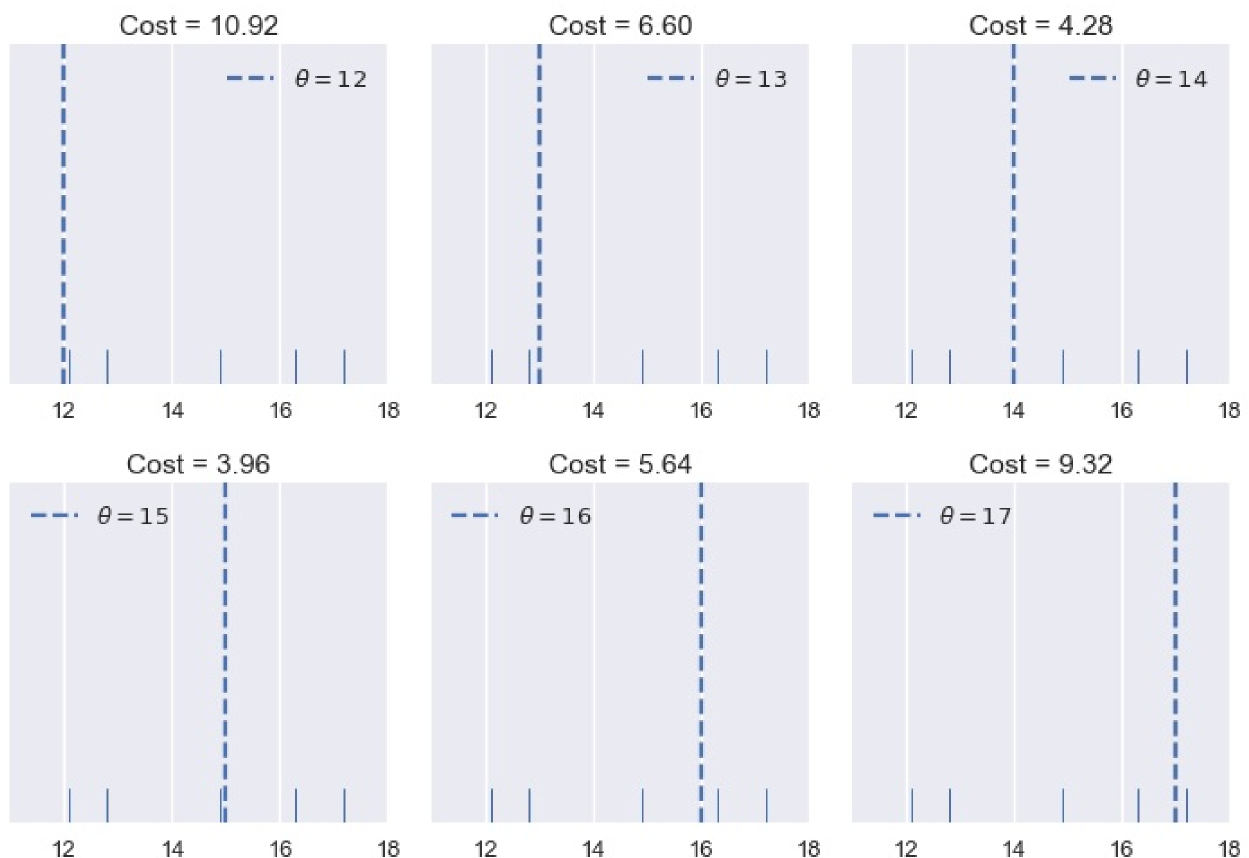
As a shorthand, we can define the vector $ y = [y_1, y_2, \ldots, y_n] $. Then, we can write our MSE cost as:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \end{aligned} $$

Finding the Best Value of $ \theta $¶

So far, we have found the best value of $ \theta $ by simply trying out a bunch of values and then picking the one with the least cost. Although this method works decently well, we can find a better method by using the properties of our cost function.

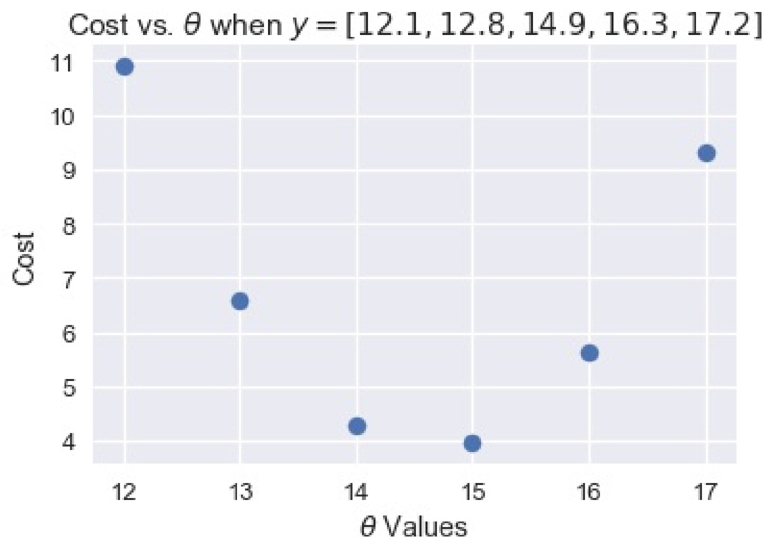
For the time being, let's return to our example with five points: $ y = [12.1, 12.8, 14.9, 16.3, 17.2] $.



In the plots above, we've used integer $ \theta $ values in between 12 and 17. When we change $ \theta $, the cost seems to start high (at 10.92), decrease until $ \theta = 15 $, then increase again. We can see that the cost changes as $ \theta $ changes, so let's make a plot comparing the cost to $ \theta $ for each of the six $ \theta $s we've tried.

```
thetas = np.array([12, 13, 14, 15, 16, 17])
y_vals = np.array([12.1, 12.8, 14.9, 16.3, 17.2])
costs = [mse_cost(theta, y_vals) for theta in thetas]

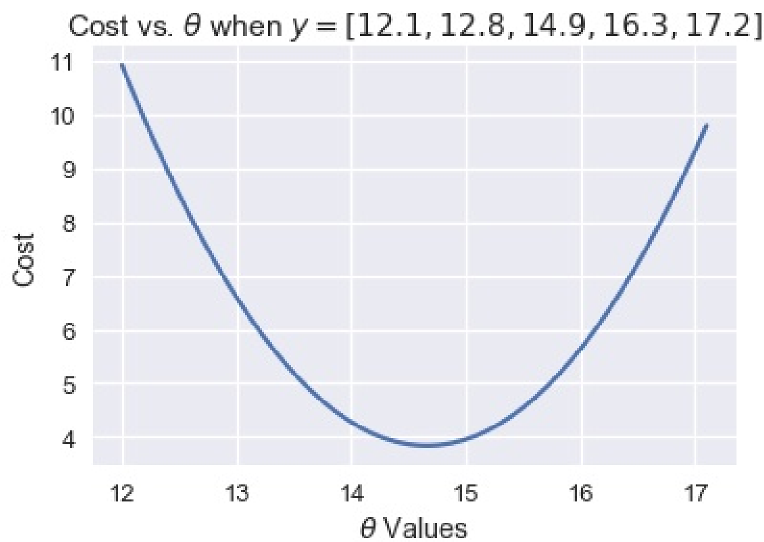
plt.scatter(thetas, costs)
plt.title(r'Cost vs. $ \theta $ when $ y = [ 12.1, 12.8, 14.9,
16.3, 17.2 ] $')
plt.xlabel(r'$ \theta $ Values')
plt.ylabel('Cost');
```



The scatter plot shows the downward, then upward trend that we noticed before. We can try more values of $ \theta $ to see a complete curve that shows how the cost changes as $ \theta $ changes.

```
thetas = np.arange(12, 17.1, 0.05)
y_vals = np.array([12.1, 12.8, 14.9, 16.3, 17.2])
costs = [mse_cost(theta, y_vals) for theta in thetas]

plt.plot(thetas, costs)
plt.title(r'Cost vs. $ \theta $ when $ y = [ 12.1, 12.8, 14.9,
16.3, 17.2 ] $')
plt.xlabel(r'$ \theta $ Values')
plt.ylabel('Cost');
```



The plot above shows that in fact, $ \theta = 15$ was not the best choice; a $ \theta $ of around 14.7 would have gotten a lower cost! We can use calculus to find that minimizing value of $ \theta $ exactly. First, we start with our cost function:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \end{aligned} $$

And plug in our points $ y = [12.1, 12.8, 14.9, 16.3, 17.2] $:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \big((12.1 - \theta)^2 + (12.8 - \theta)^2 + (14.9 - \theta)^2 + (16.3 - \theta)^2 + (17.2 - \theta)^2 \big)\\ \end{aligned} $$

To find the value of $ \theta $ that minimizes this function, we compute the derivative with respect to $ \theta $:

$$ \begin{aligned} \frac{\partial}{\partial \theta} L(\theta, y) &= \frac{1}{n} \big(-2(12.1 - \theta) - 2(12.8 - \theta) - 2(14.9 - \theta) - 2(16.3 - \theta) -2(17.2 - \theta) \big)\\ &= - \frac{2}{n} \big((12.1 - \theta) + (12.8 - \theta) + (14.9 - \theta) + (16.3 - \theta) + (17.2 - \theta) \big) \end{aligned} $$

Then, we find the value of $ \theta $ where the derivative is zero:

$$ \begin{aligned} 0 &= - \frac{2}{n} \big((12.1 - \theta) + (12.8 - \theta) + (14.9 - \theta) + (16.3 - \theta) + (17.2 - \theta) \big) \\ 0 &= (12.1 - \theta) + (12.8 - \theta) + (14.9 - \theta) + (16.3 - \theta) + (17.2 - \theta) \\ 5 \theta &= 12.1 + 12.8 + 14.9 + 16.3 + 17.2 \\ \theta &= \frac{12.1 + 12.8 + 14.9 + 16.3 + 17.2}{5} \\ \theta &= 14.66 \end{aligned} $$

As expected, the value of $ \theta $ that minimizes the cost is between 14 and 15.

Turn your attention to the second-to-last line of the simplification above:

$$ \theta = \frac{12.1 + 12.8 + 14.9 + 16.3 + 17.2}{5} $$

Notice that this is a familiar expression: it is the average of the five data points. Could this be a pattern for all values of $ y $?

The Minimizing Value of the Mean Squared Error¶

We have seen that different values of θ produce different costs when using the mean squared error cost function. The arithmetic above hints that the value of θ that minimizes the cost is the mean of all of the data points. To confirm this, we turn back to the definition of our cost function. Instead of plugging in points, we take the derivative with respect to θ of the cost function as-is:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \frac{\partial}{\partial \theta} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n} -2(y_i - \theta) \\ &= -\frac{2}{n} \sum_{i = 1}^{n} (y_i - \theta) \\ \end{aligned} $$

Notice that we have left the variables y_i untouched. We are no longer working with the previous example dataset of five points; this equation can be used with any dataset with any number of points.

Now, we set the derivative equal to zero and solve for θ to find the minimizing value of θ as before:

$$ \begin{aligned} -\frac{2}{n} \sum_{i = 1}^{n} (y_i - \theta) &= 0 \\ \sum_{i = 1}^{n} (y_i - \theta) &= 0 \\ \sum_{i = 1}^{n} y_i - \sum_{i = 1}^{n} \theta &= 0 \\ \sum_{i = 1}^{n} \theta &= \sum_{i = 1}^{n} y_i \\ n \cdot \theta &= y_1 + \ldots + y_n \\ \theta &= \frac{y_1 + \ldots + y_n}{n} \\ \theta &= \text{mean} (y) \end{aligned} $$

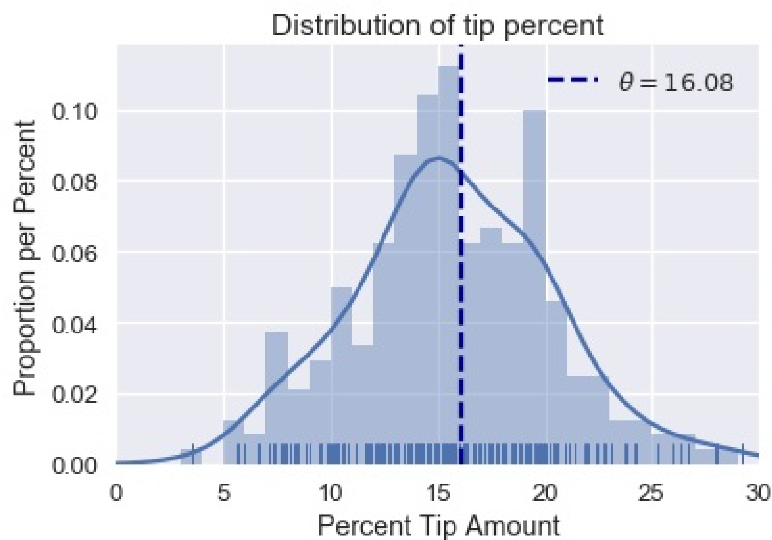
Lo and behold, we see that there is a single value of θ that gives the least MSE no matter what the dataset is. For the mean squared error, we can set θ equal to the mean of the dataset and be confident knowing that we have minimized the cost.

Back to the Original Dataset¶

We no longer have to test out different values of θ as we did before. We can compute the mean tip percentage in one go:

```
np.mean(tips['pcttip'])
```

```
16.080258172250463
```



Summary¶

First, we restricted our model to only make a single number as its prediction for all tables. Next, we assume that the waiter's dataset of tips is similar to the population distribution of tip percentages. If this assumption holds, predicting $ 16.08\% $ will give us the most accurate predictions that we can given our data.

To be more precise, we say that the model is accurate if it minimizes the squared difference between the predictions and the actual values.

Although our model is simple, it illustrates concepts that we'll see over and over again. Future chapters will introduce complicated models. Still, we will discuss each model's assumptions, define cost functions, and find the model that minimizes the cost. It is very helpful to understand this process for simple models before attempting to understand complex ones.

Show Widgets Open on DataHub

Table of Contents

Absolute and Huber Cost¶

Previously, we said that our model is accurate if it minimizes the squared difference between the predictions and the actual values. We used the mean squared error (MSE) cost to capture this measure of accuracy:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \end{aligned} $$

We used a simple model that always predicts the same number:

$$ \theta = C $$

Where C is some constant. When we use this constant model and the MSE cost function, we found that C will always be the mean of the data points. When applied to the tips dataset, we found that the constant model should predict 16.08% since 16.08% is the mean of the tip percents.

Now, we will keep our model the same but switch to a different cost function: the mean absolute cost. Instead taking the squared difference for each point and our prediction, this cost function takes the absolute difference:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n} |y_i - \theta| \\ \end{aligned} $$

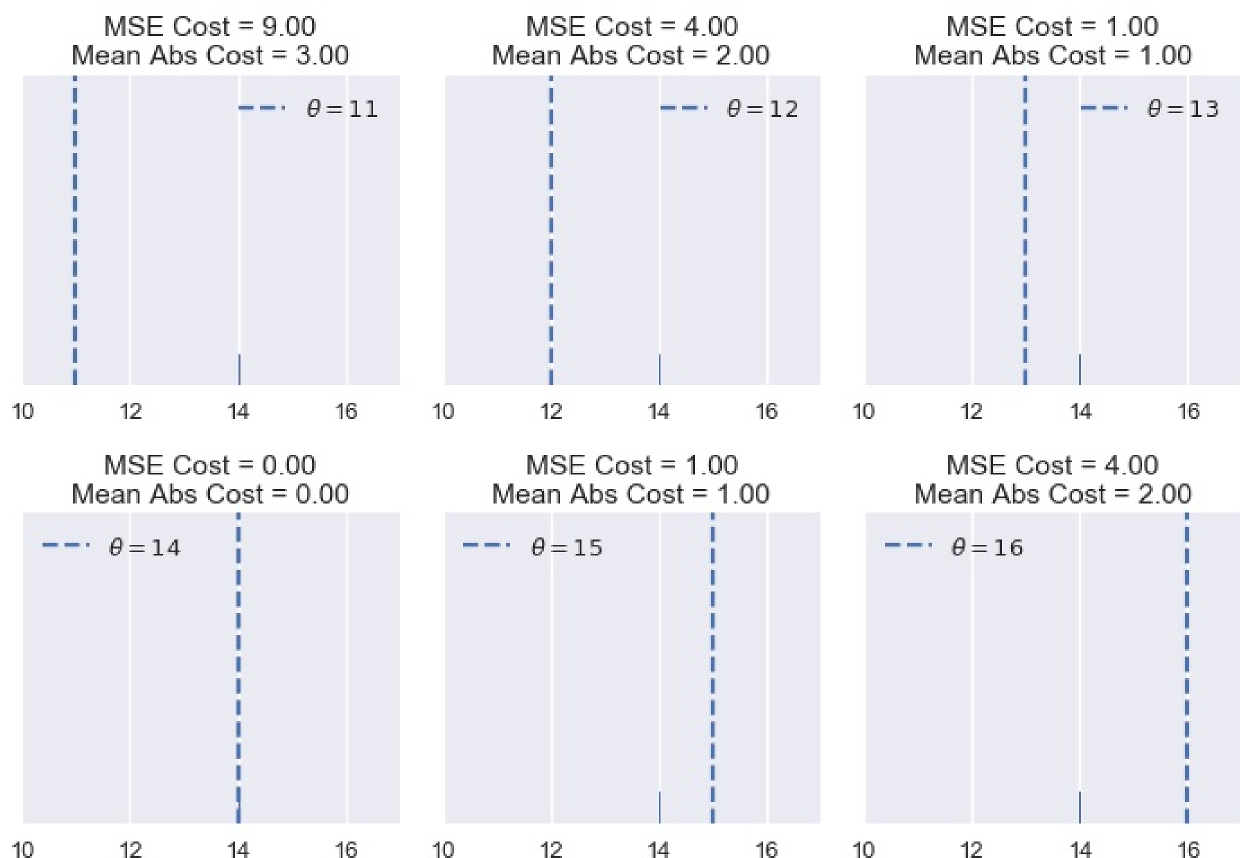
Brief Aside: Cost Functions vs. Loss Functions

We typically use the term *cost function* to describe a function that takes in parameters of our model (e.g. θ) and the entire set of data that we are interested in (e.g. y). This function generates a single value that describes how well the model does on a given set of points.

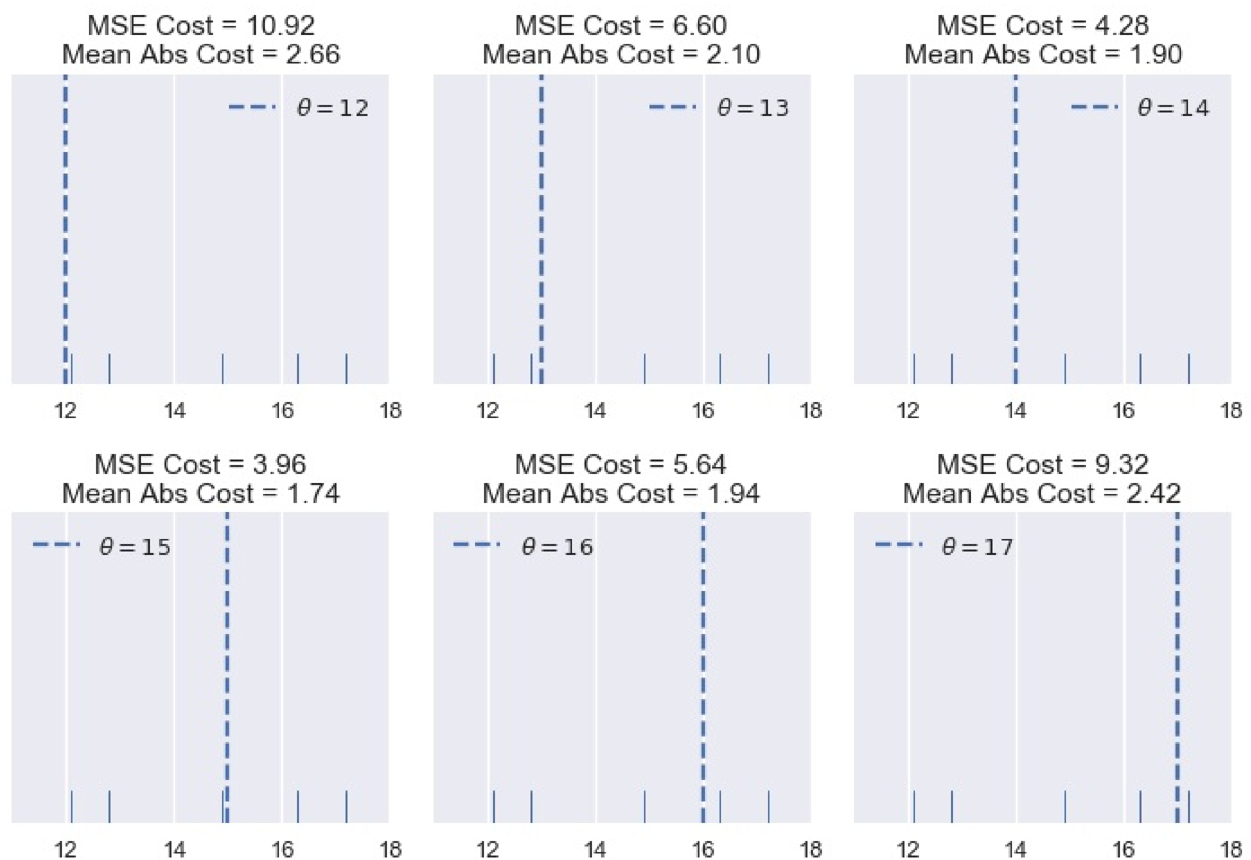
We typically use the term *loss function* to describe a function that takes in parameters of our model (e.g. $ \theta $) and one point from the data that we are interested in (e.g. $ y_i $). For example, $ (y_i - \theta)^2 $ is the squared loss, and $ |y_i - \theta| $ is the absolute loss. Notice that there is no summation in these example loss functions. You can think of a cost function as the combination of multiple loss function values.

Comparing MSE and Mean Absolute Cost¶

To get a better sense of how the MSE cost and mean absolute cost compare, let's compare their costs on different datasets. First, we'll use our dataset of one point: $ y = [14] $.

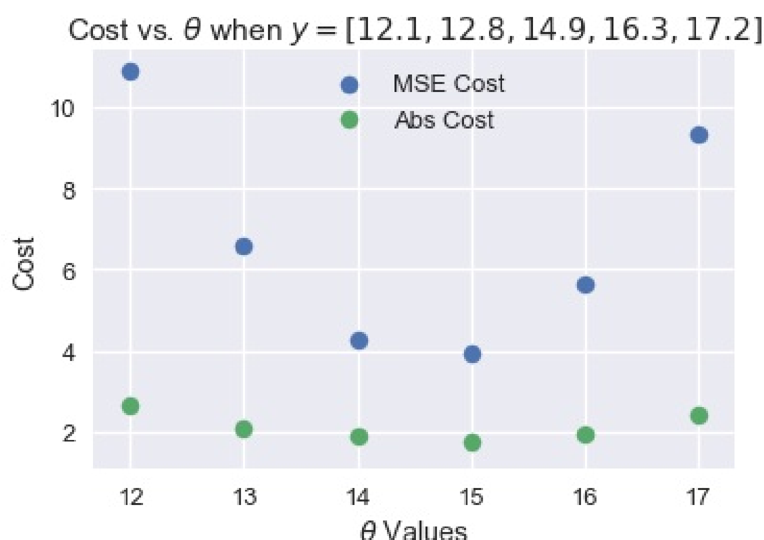


We see that the MSE cost is usually higher than the mean absolute cost since the error is squared. Let's see what happens when have five points: $ y = [12.1, 12.8, 14.9, 16.3, 17.2] $

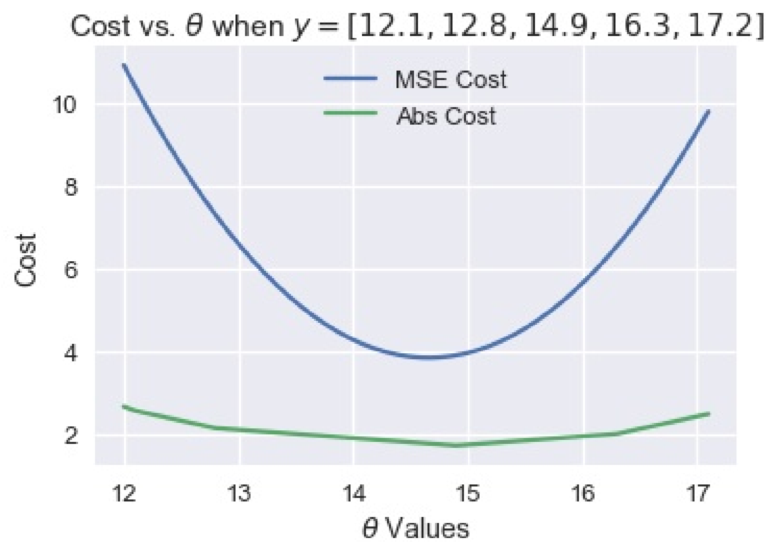


Remember that the actual cost values themselves are not very interesting to us; they are only useful for comparing different values of $ \theta $. Once we choose a cost function, we will look for the $ \theta $ that produces the least cost. Thus, we are interested in whether the cost functions are minimized by different values of $ \theta $.

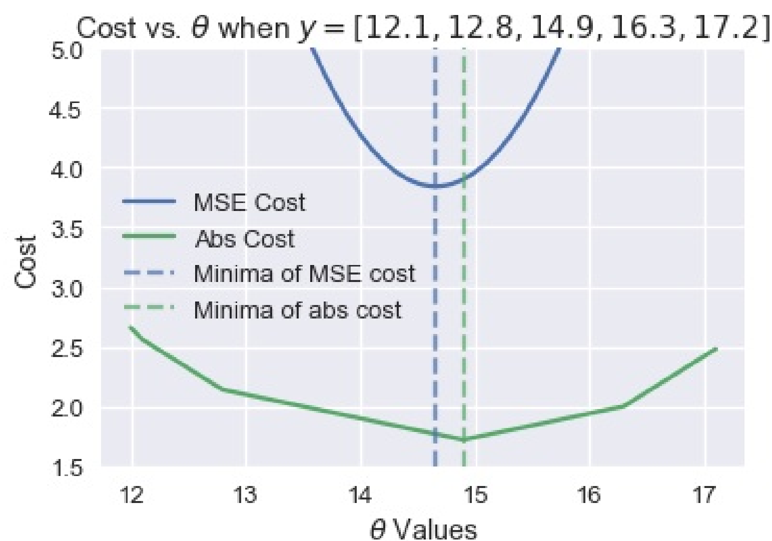
So far, the two cost functions seem to agree on the $ \theta $ values that produce the least cost for the values that we've tried. If we look a bit closer, however, we will start to see some differences. We first take the costs and plot them against $ \theta $ for each of the six $ \theta $ values we tried.



Then, we compute more values of $ \theta $ so that the curve is smooth:



Then, we zoom into the region between 1.5 and 5 on the y-axis to see the difference in minima more clearly. We've marked the minima with dotted lines.



We've found empirically that the MSE cost and mean absolute cost can be minimized by different values of $ \theta $ for the same dataset. However, we don't have a good sense of when they will differ and more importantly, why they differ.

Outliers¶

One difference that we can see in the plots of cost vs. $ \theta $ above lies in the shape of the cost curves. Plotting the MSE cost results in a parabolic curve resulting from the squared term in the cost function.

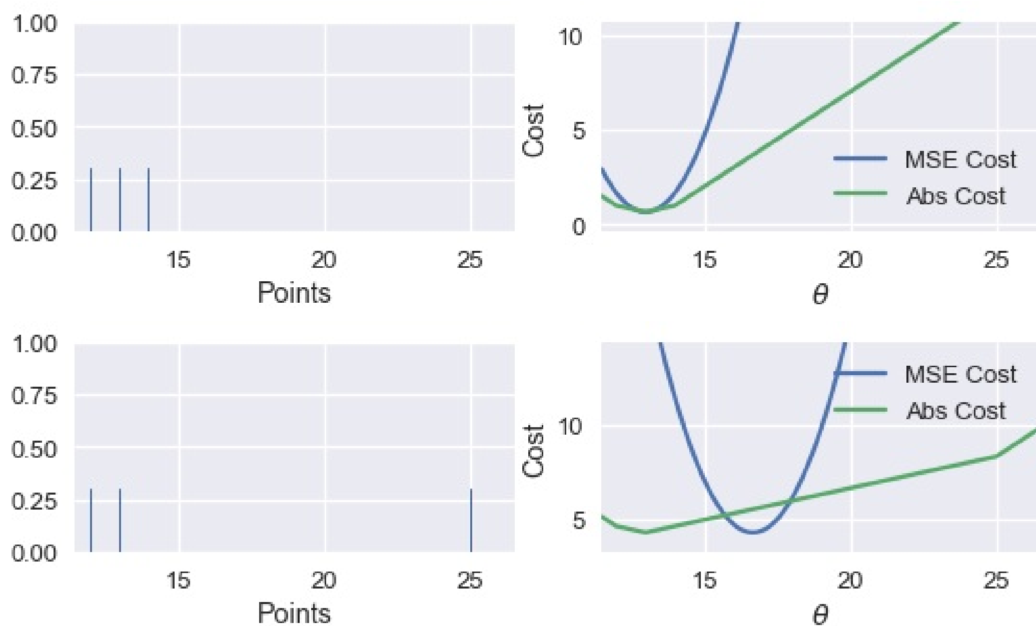
Plotting the mean absolute cost, on the other hand, results in what looks like a connected series of lines. This makes sense when we consider that the absolute value function is linear, so taking the average of many absolute value functions should produce a semi-linear function.

Since the MSE cost has a squared error term, it will be more sensitive to outliers. If $ \theta = 10 $ and a point lies at 110, that point's error term for MSE will be $ (10 - 110)^2 = 10000 $ whereas in the mean absolute cost, that point's error term will be $ |10 - 110| = 100 $. We can illustrate this by taking a set of three points $ y = [12, 13, 14] $ and plotting the cost vs. $ \theta $ curves for MSE and mean absolute loss.

Use the slider below to move the third point further away from the rest of the data and observe what happens to the cost curves. (We've scaled the curves to keep both in view since the MSE cost has larger values than the mean absolute cost.)

Show Widget

We've shown the curves for $ y_3 = 14 $ and $ y_3 = 25 $ below.



As we move the point further away from the rest of the data, the MSE cost curve moves with it. When $ y_3 = 14 $, both MSE and mean absolute cost will be minimized by the same value of $ \theta $ at $ \theta = 13 $. However, when $ y_3 = 25 $, the MSE curve will tell us that the best $ \theta $ is around 16.7 while the mean absolute cost will still say that $ \theta = 13 $ is best.

Minimizing the Mean Absolute Loss¶

Now that we have a qualitative sense of how the MSE and mean absolute cost differ, we can minimize the mean absolute cost to make this difference more precise. As before, we will take the derivative of the cost function with respect to $ \theta $ and set it equal to zero.

This time, however, we have to deal with the fact that the absolute function is not always differentiable. When $ x > 0 $, $ \frac{\partial}{\partial x} |x| = 1 $. When $ x < 0 $, $ \frac{\partial}{\partial x} |x| = -1 $. Although $ |x| $ is not technicaly differentiable at $ x = 0 $, we will set $ \frac{\partial}{\partial x} |x| = 0 $ so that the equations are easier to work with.

Recall that the equation for the mean absolute loss is:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}|y_i - \theta|\\ &= \frac{1}{n} \left(\sum_{y_i < \theta}|y_i - \theta| + \sum_{y_i = \theta}|y_i - \theta| + \sum_{y_i > \theta}|y_i - \theta| \right)\\ \end{aligned} $$

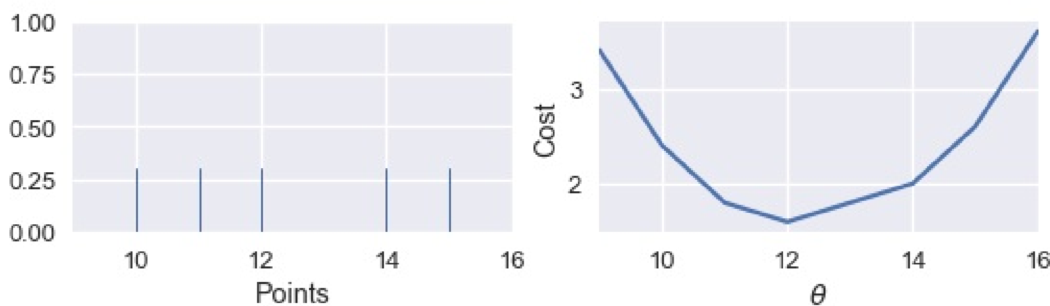
In the line above, we've split up the summation into three separate summations: one that has one term for each $ y_i < \theta $, one for $ y_i = \theta $, and one for $ y_i > \theta $. Why make the summation seemingly more complicated? If we know that $ y_i < \theta $ we also know that $ |y_i - \theta| < 0 $ and thus $ \frac{\partial}{\partial \theta} |y_i - \theta| = -1 $ from before. A similar logic holds for each term above to make taking the derivative much easier.

Now, we take the derivative with respect to $ \theta $ and set it equal to zero:

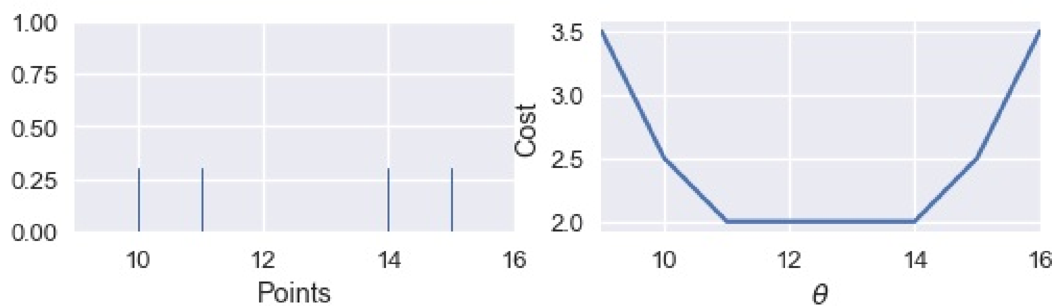
$$ \begin{aligned} \frac{1}{n} \left(\sum_{y_i < \theta}(-1) + \sum_{y_i = \theta}(0) + \sum_{y_i > \theta}(1) \right) &= 0 \\ \sum_{y_i < \theta}(-1) + \sum_{y_i > \theta}(1) &= 0 \\ -\sum_{y_i < \theta}(1) + \sum_{y_i > \theta}(1) &= 0 \\ \sum_{y_i < \theta}(1) &= \sum_{y_i > \theta}(1) \\ \end{aligned} $$

What does the result above mean? On the left hand side, we have one term for each data point less than $ \theta $. On the right, we have one for each data point greater than $ \theta $. Then, in order to satisfy the equation we need to pick a value for $ \theta $ that has the same number of smaller and larger points. This is the definition for the *median* of a set of numbers. Thus, the minimizing value of $ \theta $ for the mean absolute cost is $ \theta = \text{median} (y) $.

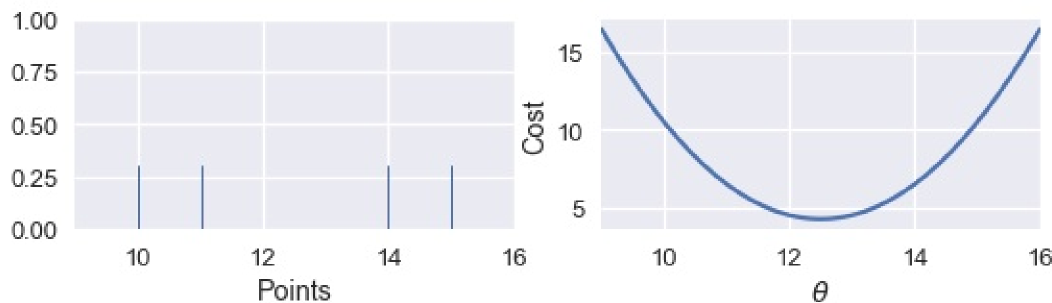
When we have an odd number of points, the median is simply the middle point when the points are arranged in sorted order. We can see that in the example below with five points, the cost is minimized when $ \theta $ lies at the median:



However, when we have an even number of points, the cost is minimized when $ \theta $ is any value in between the two central points.



This is not the case when we use the MSE cost:



MSE and Mean Absolute Cost Comparison¶

Our investigation and the derivation above show that the MSE is easier to differentiate but is more sensitive to outliers than the mean absolute cost. The minimizing $ \theta $ for MSE is the mean of the data points, and the minimizing $ \theta $ for the mean absolute cost is the median of the data points. Notice that the median is robust to outliers while the mean is not! This phenomenon arises from our construction of the two cost functions.

We have also seen that the MSE cost will be minimized by a unique value of $ \theta $, whereas the mean absolute value can be minimized by multiple values of $ \theta $ when there are an even number of data points.

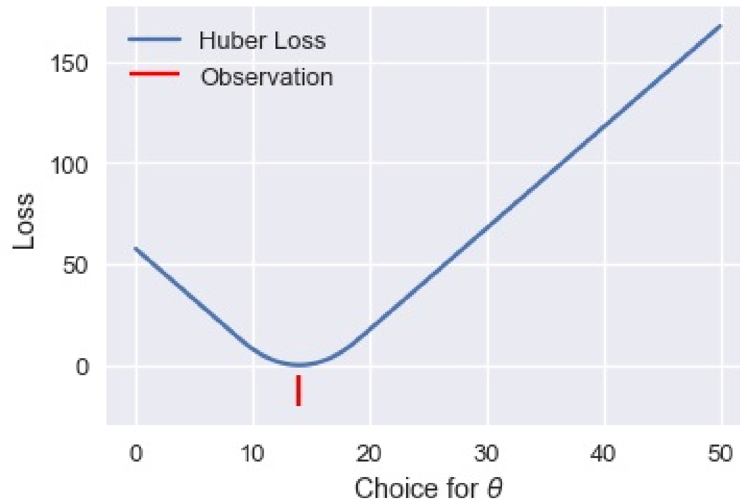
In the examples so far, the ability to differentiate the cost function isn't that useful since we know the exact minimizing value of $ \theta $ in both cases. However, the ability to differentiate the cost function becomes very important once we start using complicated models. For complicated models, we will not be able to differentiate the cost function by hand and will need a computer to minimize the cost function for us. We will return to this issue when we cover gradient descent and numerical optimization.

The Huber Cost¶

A third loss function called the Huber loss combines both the MSE and mean absolute loss to create a loss function that is differentiable *and* robust to outliers. The Huber loss accomplishes this by behaving like the MSE loss function at values close to $ \theta $ and switching to the absolute loss for values far from $ \theta $.

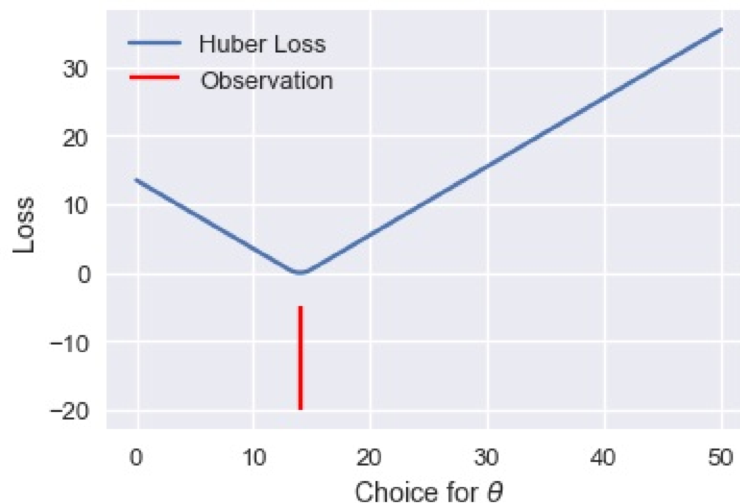
As usual, we create a cost function by taking the mean of the Huber losses for each point in our dataset.

Let's see what the Huber cost function outputs for a dataset of $ y = [14] $ as we vary $ \theta $:

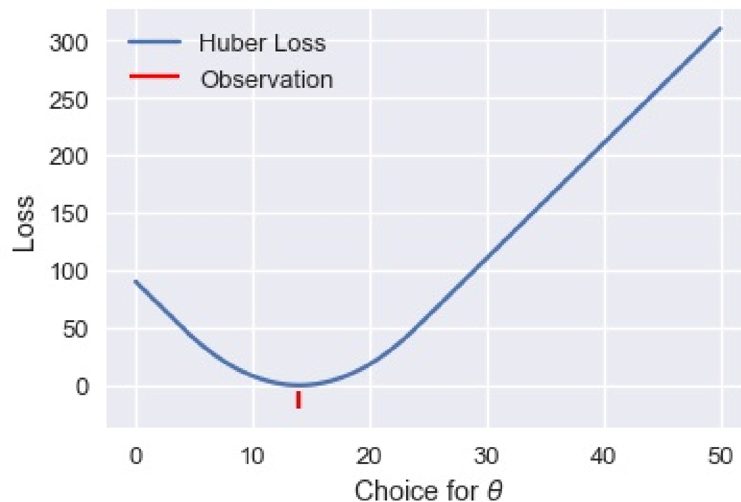


We can see that the Huber cost is smooth, unlike the mean absolute cost. The Huber cost also increases at a linear rate, unlike the quadratic rate of the mean squared cost.

The Huber cost does have a drawback, however. Notice that it transitions from the MSE to the mean absolute cost once $ \theta $ gets far enough from the point. We can tweak this "far enough" to get different cost curves. For example, we can make it transition once $ \theta $ is just one unit away from the observation:



Or we can make it transition when $ \theta $ is ten units away from the observation:



This choice results in a different cost curve and can thus result in different values of $ \theta $. If we want to use the Huber cost function, we have the additional task of setting this transition point to a suitable value.

The Huber cost function is defined mathematically as follows:

$$ L_\alpha(\theta, y) = \frac{1}{n} \sum_{i=1}^n \begin{cases} \frac{1}{2}(y_i - \theta)^2 & | y_i - \theta | \le \alpha \\ \alpha (|y_i - \theta| - \frac{1}{2}\alpha) & \text{otherwise} \end{cases} $$

It is more complex than the previous cost functions because it combines both MSE and mean absolute cost. The additional parameter $ \alpha $ sets the point where the Huber loss transitions from the MSE cost to the absolute cost.

Attempting to take the derivative of the Huber loss function is tedious and does not result in an elegant result like the MSE and mean absolute loss. Instead, we can use a computational method called gradient descent to find minimizing value of $ \theta $.

Gradient Descent and Numerical Optimization

In order to use a dataset for estimation and prediction, we need to precisely define our model and select a cost function. For example, in the tip percentage dataset, our model assumed that there was a single tip percentage that does not vary by table. Then, we decided to use the mean squared error cost function and found the model that minimized the cost function.

We also found that there are simple expressions that minimize the MSE and the mean absolute cost functions: the mean and the median. However, as our models and cost functions become more complicated we will no longer be able to find useful algebraic expressions for the models that minimize the cost. For example, the Huber cost has useful properties but is difficult to differentiate by hand.

We can use the computer to address this issue using gradient descent, a computational method of minimizing cost functions.

Show Widgets Open on DataHub

Table of Contents

Cost Minimization Using a Program¶

Let us return to our constant model:

$$ \theta = C $$

We will use the mean squared error cost function:

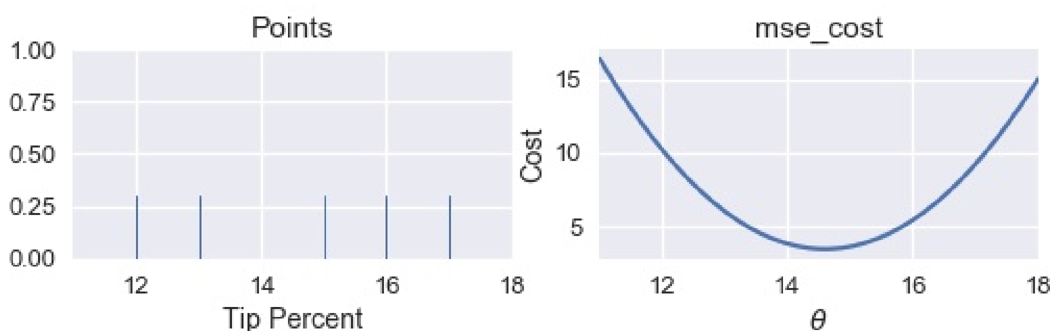
$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \end{aligned} $$

For simplicity, we will use the dataset $ y = [12, 13, 15, 16, 17] $. We know from our analytical approach in a previous chapter that the minimizing $ \theta $ for the MSE cost is $ \text{mean}(y) = 14.6 $. Let's see whether we can find the same value by writing a program.

If we write the program well, we will be able to use the same program on any cost function in order to find the minimizing value of $ \theta $, including the mathematically complicated Huber cost:

$$ L_\alpha(\theta, y) = \frac{1}{n} \sum_{i=1}^n \begin{cases} \frac{1}{2}(y_i - \theta)^2 & | y_i - \theta | \le \alpha \\ \alpha (|y_i - \theta| - \frac{1}{2}\alpha) & \text{otherwise} \end{cases} $$

First, we create a rug plot of the data points. To the right of the rug plot we plot the MSE cost for different values of $ \theta $.



How might we write a program to automatically find the minimizing value of $ \theta $? The simplest method is to compute the cost for many values $ \theta $. Then, we can return the $ \theta $ value that resulted in the least cost.

We define a function called `simple_minimize` that takes in a cost function, an array of data points, and an array of θ values to try.

```
def simple_minimize(cost_fn, dataset, thetas):
    '''
    Returns the value of theta in thetas that produces the least
cost
    on a given dataset.
    '''
    costs = [cost_fn(theta, dataset) for theta in thetas]
    return thetas[np.argmin(costs)]
```

Then, we can define a function to compute the MSE cost and pass it into `simple_minimize`.

```
def mse_cost(theta, dataset):
    return np.mean((dataset - theta) ** 2)

dataset = np.array([12, 13, 15, 16, 17])
thetas = np.arange(12, 18, 0.1)

simple_minimize(mse_cost, dataset, thetas)
```

```
14.599999999999991
```

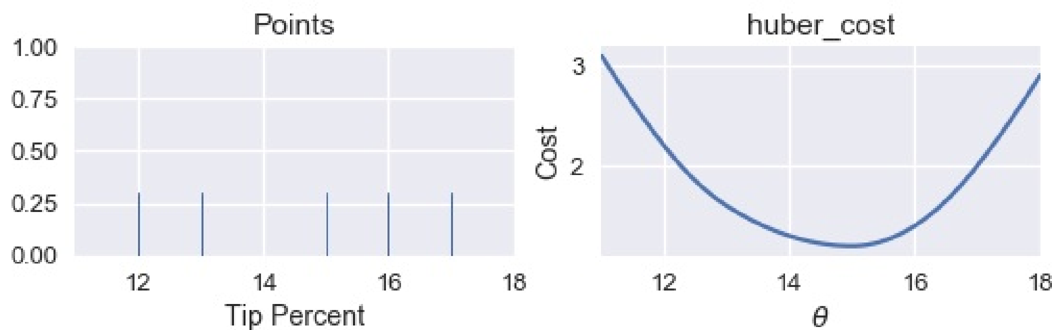
This is close to the expected value:

```
# Compute the minimizing theta using the analytical formula
np.mean(dataset)
```

```
14.6
```

Now, we can define a function to compute the Huber cost and plot the cost against θ.

```
def huber_cost(theta, dataset, alpha = 1):
    d = np.abs(theta - dataset)
    return np.mean(
        np.where(d < alpha,
                 (theta - dataset)**2 / 2.0,
                 alpha * (d - alpha / 2.0))
    )
```



Although we can see that the minimizing value of $ \theta $ should be close to 15, we do not have an analytical method of finding $ \theta $ directly for the Huber cost. Instead, we can use our `simple_minimize` function.

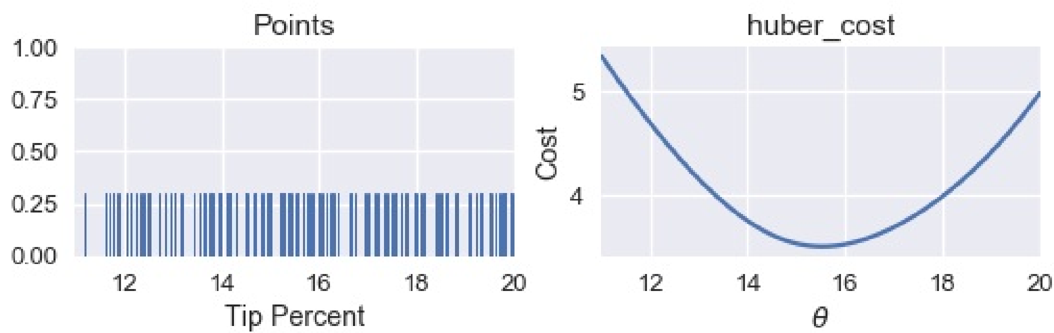
```
simple_minimize(huber_cost, dataset, thetas)
```

```
14.999999999999989
```

Now, we can return to our original dataset of tip percentages and find the best value for $ \theta $ using the Huber cost.

```
tips = sns.load_dataset('tips')
tips['pcttip'] = tips['tip'] / tips['total_bill'] * 100
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size	pcttip
0	16.99	1.01	Female	No	Sun	Dinner	2	5.944673
1	10.34	1.66	Male	No	Sun	Dinner	3	16.054159
2	21.01	3.50	Male	No	Sun	Dinner	3	16.658734
3	23.68	3.31	Male	No	Sun	Dinner	2	13.978041
4	24.59	3.61	Female	No	Sun	Dinner	4	14.680765




```
simple_minimize(huber_cost, tips['pcttip'], thetas)
```

```
15.499999999999988
```

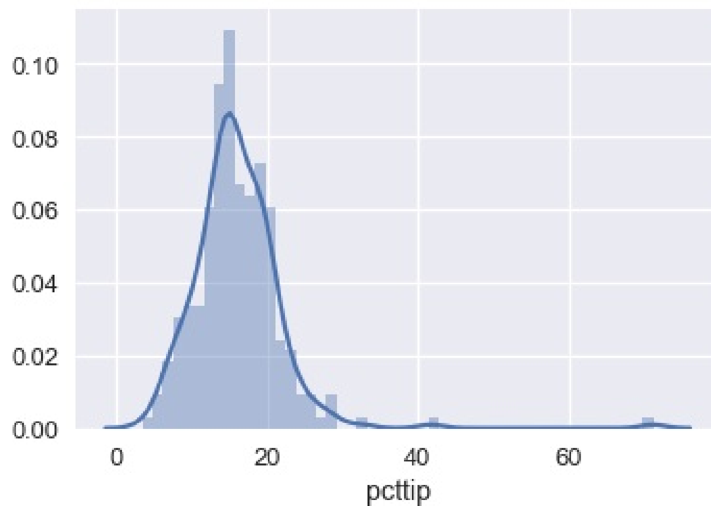
We can see that using the Huber cost gives us $ \theta = 15.5 $. We can now compare the minimizing $ \theta $ values for MSE cost, mean absolute cost, and Huber cost.

```
print(f"          MSE cost: theta =
{tips['pcttip'].mean():.2f}")
print(f"Mean Absolute cost: theta =
{tips['pcttip'].median():.2f}")
print(f"        Huber cost: theta = 15.50")
```

```
          MSE cost: theta = 16.08
Mean Absolute cost: theta = 15.48
        Huber cost: theta = 15.50
```

We can see that the Huber cost is closer to the mean absolute cost since it is less affected by the outliers on the right side of the tip percentage distribution:

```
sns.distplot(tips['pcttip'], bins=50);
```



Issues with `simple_minimize` ¶

Although `simple_minimize` allows us to minimize cost functions, it has some flaws that make it unsuitable for general purpose use. It's primary issue is that it only works with predetermined values of $ \theta $ to test. For example, in this code snippet we used above, we had to manually define $ \theta $ values in between 12 and 18.

```
dataset = np.array([12, 13, 15, 16, 17])
thetas = np.arange(12, 18, 0.1)

simple_minimize(mse_cost, dataset, thetas)
```

How did we know to examine the range between 12 and 18? We had to inspect the plot of the cost function manually and see that there was a minima in that range. This process becomes impractical as we add extra complexity to our models. In addition, we manually specified a step size of 0.1 in the code above. However, if the optimal value of $ \theta $ was 12.043, our `simple_minimize` function would round to 12.00, the nearest multiple of 0.1.

We can solve both of these issues at once by using a method called *gradient descent*.

Show Widgets Open on DataHub

Table of Contents

Gradient Descent¶

We are interested in creating a function that can minimize a cost function without forcing the user to predetermine which values of θ to try. In other words, while the `simple_minimize` function has the following signature:

```
simple_minimize(cost_fn, dataset, thetas)
```

We would like a function that has the following signature:

```
minimize(cost_fn, dataset)
```

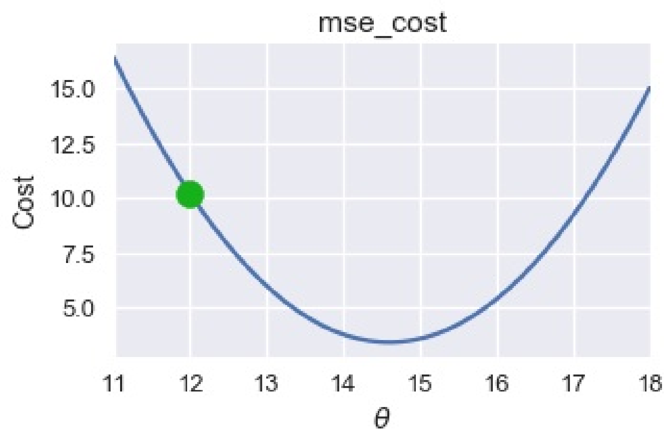
This function needs to automatically find the minimizing θ value no matter how small or large it is. We will use a technique called gradient descent to implement this new `minimize` function.

Intuition¶

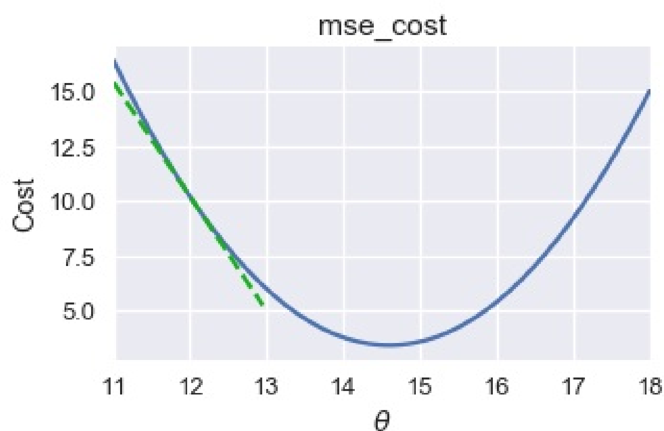
As with cost functions, we will discuss the intuition for gradient descent first, then formalize our understanding with mathematics.

Since the `minimize` function isn't given values of θ to try, we start by picking a θ anywhere we'd like. Then, we can iteratively improve the estimate of θ. To improve an estimate of θ, we look at the slope of the cost function at that choice of θ.

For example, suppose we are using MSE cost for the simple dataset $y = [12, 13, 15, 16, 17]$ and our current choice of θ is 12.

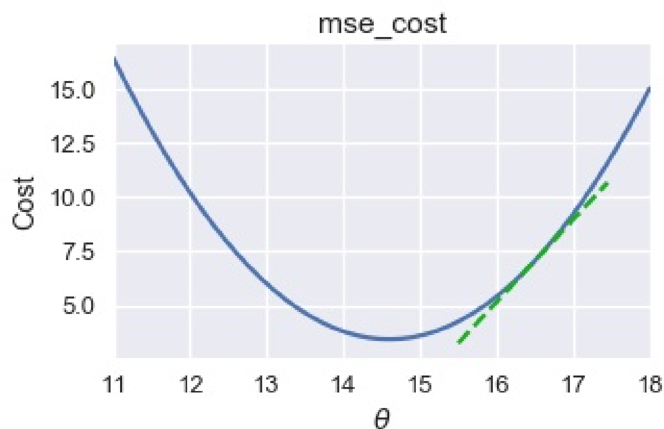


We'd like to choose a new value for θ that decreases the cost. To do this, we look at the slope of the cost function at $\theta = 12$:



The slope is negative, which means that increasing θ will decrease the cost.

If $\theta = 16.5$ on the other hand, the slope of the cost function is positive:



When the slope is positive, decreasing θ will decrease the cost.

The slope of the tangent line tells us which direction to move θ in order to decrease the cost. If the slope is negative, we want θ to move in the positive direction. If the slope is positive, θ should move in the negative direction. Mathematically, we write:

$$\theta_{t+1} = \theta_t - \frac{\partial}{\partial \theta} L(\theta, y)$$

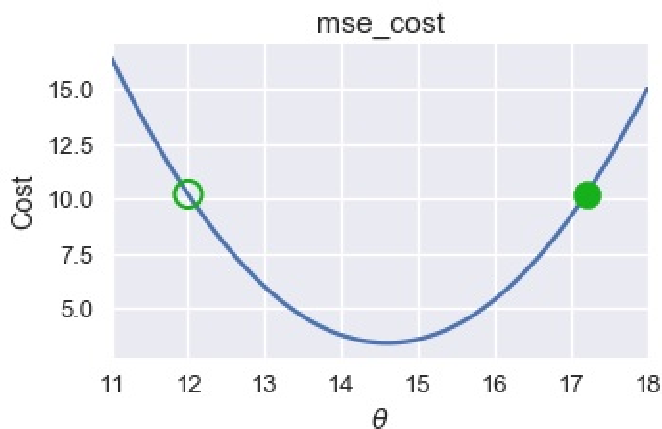
Where $ \theta_t $ is the current estimate and $ \theta_{t+1} $ is the next estimate.

For the MSE cost, we have:

$$ \begin{aligned} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - \theta)^2\\ \frac{\partial}{\partial \theta} L(\theta, y) &= \frac{1}{n} \sum_{i = 1}^{n} -2(y_i - \theta) \\ &= -\frac{2}{n} \sum_{i = 1}^{n} (y_i - \theta) \\ \end{aligned} $$

When $ \theta_t = 12 $, we can compute $ -\frac{2}{n} \sum_{i = 1}^{n} (y_i - \theta) = -5.2 $. Thus, $ \theta_{t+1} = 12 - (-5.2) = 17.2 $.

We've plotted the old value of $ \theta $ as a green outlined circle and the new value as a filled in circle on the cost curve below.

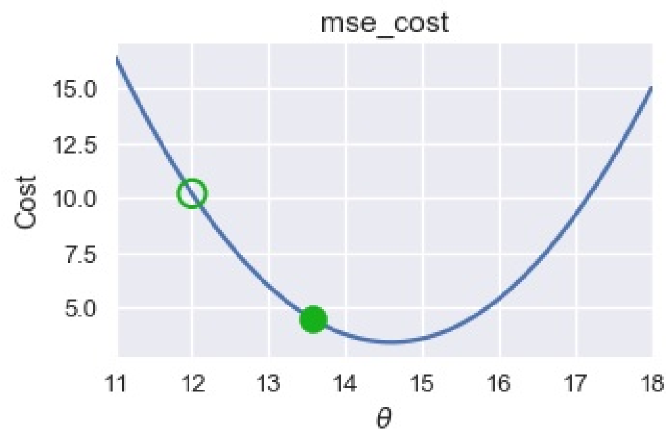


Although $ \theta $ went in the right direction, it ended up as far away from the minimum as it started. We can remedy this by multiplying the slope by a small constant before subtracting it from $ \theta $. Our final update formula is:

$$ \theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta} L(\theta, y) $$

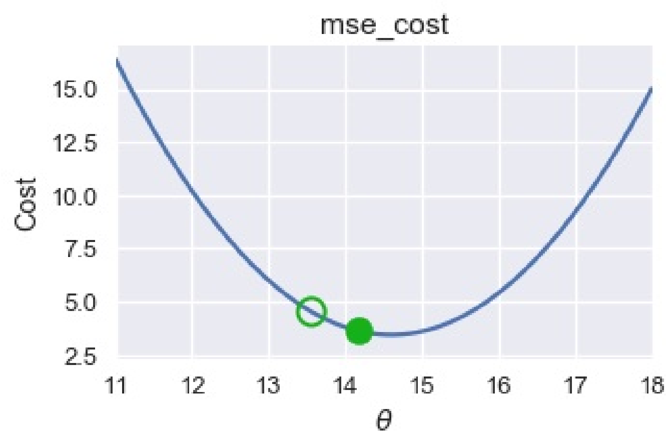
Where $ \alpha $ is a small constant. For example, if we set $ \alpha = 0.3 $ this is the new $ \theta_{t+1} $:

```
old theta: 12
new theta: 13.56
```

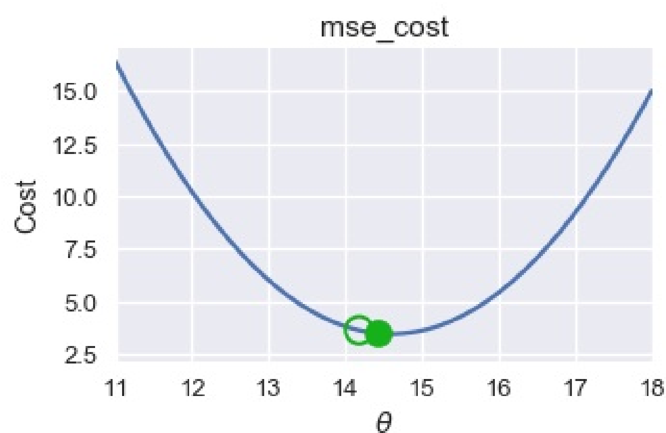


Here are the $ \theta $ values for successive iterations of this process. Notice that $ \theta $ changes more slowly as it gets closer to the minimum cost because the slope is also smaller.

```
old theta: 13.56
new theta: 14.184000000000001
```




```
old theta: 14.18
new theta: 14.432
```



```
old theta: 14.432
new theta: 14.5328
```



Gradient Descent Analysis¶

We now have the full algorithm for gradient descent:

1. Choose a starting value of $ \theta $ (0 is a common choice).
2. Compute $ \theta - \alpha \cdot \frac{\partial}{\partial \theta} L(\theta, y) $ and store this as the new value of $ \theta $.
3. Repeat until $ \theta $ doesn't change between iterations.

You will more commonly see the gradient $ \nabla_\theta $ in place of the partial derivative $ \frac{\partial}{\partial \theta} $. The two notations are essentially equivalent, but since the gradient notation is more common we will use it in the gradient update formula from now on:

$$ \theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta L(\theta, y) $$

To review notation:

- $ \theta_t $ is the current choice of $ \theta $.
- $ \theta_{t+1} $ is the next choice of $ \theta $.
- $ \alpha $ is called the learning rate, usually set to a small constant. Sometimes it is useful to start with a larger $ \alpha $ and decrease it over time. If $ \alpha $ changes between iterations, we use the variable $ \alpha_t $ to mark that $ \alpha $ varies over time $ t $.
- $ \nabla_\theta L(\theta, y) $ is the partial derivative / gradient of the cost function at $ \theta $.

You can now see the importance of choosing a differentiable cost function: $ \nabla_\theta L(\theta, y) $ is a crucial part of the gradient descent algorithm. (While it is possible to estimate the gradient by computing the difference in cost for two slightly different values of $

\theta $ and dividing by the distance between $ \theta $ values, this typically increases the runtime of gradient descent so significantly that it becomes impractical to use.)

The gradient algorithm is simple yet powerful since we can use it for many types of models and many types of cost functions. It is the computational tool of choice for fitting many important models, including linear regression on large datasets and neural networks.

Defining the `minimize` Function¶

Now we return to our original task: defining the `minimize` function. We will have to change our function signature slightly since we now need to compute the gradient of the cost function.

```
def minimize(cost_fn, grad_cost_fn, dataset, alpha=0.2,
progress=True):
    '''
    Uses gradient descent to minimize cost_fn. Returns the
minimizing value of
    theta once theta changes less than 0.001 between iterations.
    '''
    theta = 0
    while True:
        if progress:
            print(f'theta: {theta:.2f} | cost: {cost_fn(theta,
dataset):.2f}')
        gradient = grad_cost_fn(theta, dataset)
        new_theta = theta - alpha * gradient

        if abs(new_theta - theta) < 0.001:
            return new_theta

        theta = new_theta
```

Then we can define functions to compute our MSE cost and its gradient:

```
def mse_cost(theta, y_vals):
    return np.mean((y_vals - theta) ** 2)

def grad_mse_cost(theta, y_vals):
    return -2 * np.mean(y_vals - theta)
```

Finally, we can use the `minimize` function to compute the minimizing value of $ \theta $ for $ y = [12, 13, 15, 16, 17] $.

```
%%time
theta = minimize(mse_cost, grad_mse_cost, np.array([12, 13, 15,
16, 17]))
print(f'Minimizing theta: {theta}')
print()
```

```
theta: 0.00 | cost: 216.60
theta: 5.84 | cost: 80.18
theta: 9.34 | cost: 31.07
theta: 11.45 | cost: 13.39
theta: 12.71 | cost: 7.02
theta: 13.46 | cost: 4.73
theta: 13.92 | cost: 3.90
theta: 14.19 | cost: 3.61
theta: 14.35 | cost: 3.50
theta: 14.45 | cost: 3.46
theta: 14.51 | cost: 3.45
theta: 14.55 | cost: 3.44
theta: 14.57 | cost: 3.44
theta: 14.58 | cost: 3.44
theta: 14.59 | cost: 3.44
theta: 14.59 | cost: 3.44
theta: 14.60 | cost: 3.44
theta: 14.60 | cost: 3.44
Minimizing theta: 14.59851722463264

CPU times: user 2.6 ms, sys: 1.6 ms, total: 4.2 ms
Wall time: 3.02 ms
```

We can see that gradient quickly finds the same solution as the analytic method:

```
np.mean([12, 13, 15, 16, 17])
```

```
14.6
```

Minimizing the Huber cost¶

Now, we can apply gradient descent to minimize the Huber cost on our dataset of tip percentages.

The Huber cost is:

$$ L_\delta(\theta, y) = \frac{1}{n} \sum_{i=1}^n \begin{cases} \frac{1}{2}(y_i - \theta)^2 & | y_i - \theta | \le \delta \\ \delta (|y_i - \theta| - \frac{1}{2} \delta) & \text{otherwise} \end{cases} $$

The gradient of the Huber cost is:

$$ \nabla*{\theta}* L\delta(\theta, y) = \frac{1}{n} \sum_{i=1}^n \begin{cases} -(y_i - \theta) & | y_i - \theta | \le \delta \

```
- \delta \cdot \text{sign} (y_i - \theta) & \text{otherwise}
```

\end{cases} $$

(Note that in previous definitions of Huber cost we used the variable $ \alpha $ to denote the transition point. To avoid confusion with the $ \alpha $ used in gradient descent, we replace the transition point parameter of the Huber loss with $ \delta $.)

```
def huber_cost(theta, dataset, delta = 1):
    d = np.abs(theta - dataset)
    return np.mean(
        np.where(d <= delta,
                 (theta - dataset)**2 / 2.0,
                 delta * (d - delta / 2.0))
    )

def grad_huber_cost(theta, dataset, delta = 1):
    d = np.abs(theta - dataset)
    return np.mean(
        np.where(d <= delta,
                 -(dataset - theta),
                 -delta * np.sign(dataset - theta))
    )
```

Let's minimize the Huber cost on the tips dataset:

```
%%time
theta = minimize(huber_cost, grad_huber_cost, tips['pcttip'],
progress=False)
print(f'Minimizing theta: {theta}')
print()
```

```
Minimizing theta: 15.506849531471964

CPU times: user 207 ms, sys: 6.9 ms, total: 213 ms
Wall time: 258 ms
```

Summary¶

Gradient descent gives us a generic way to minimize a cost function when we cannot solve for the minimizing value of θ analytically. As our models and cost functions increase in complexity, we will turn to gradient descent as our tool of choice to fit models.

Show Widgets Open on DataHub

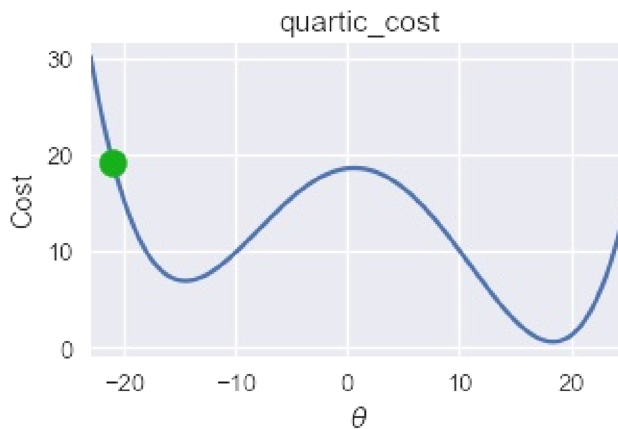
Table of Contents

Convexity¶

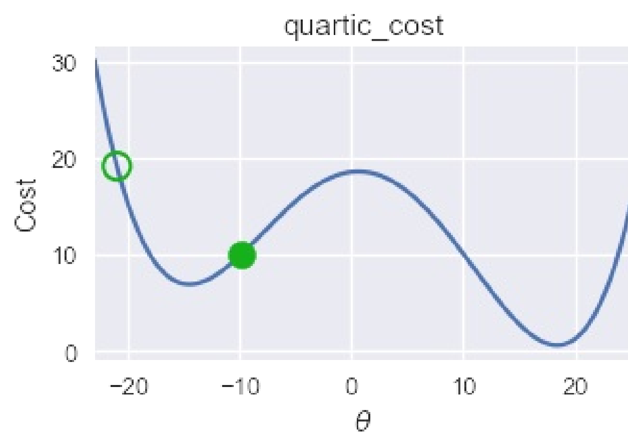
Gradient descent provides a general method for minimizing a function. As we observe for the Huber cost, gradient descent is especially useful when the function's minimum is difficult to find analytically.

Gradient Descent Finds Local Minima¶

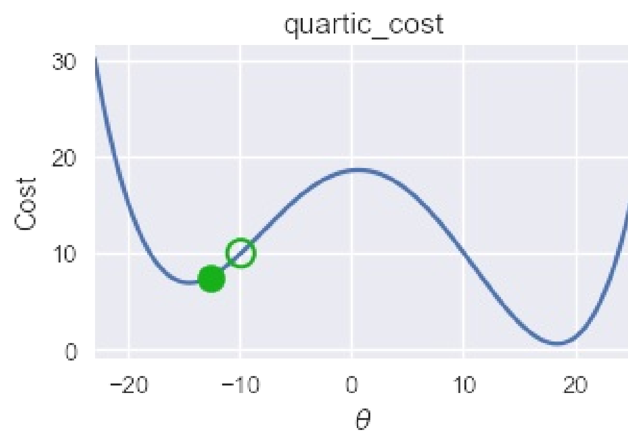
Unfortunately, gradient descent does not always find the globally minimizing $ \theta $. Consider the following gradient descent run using an initial $ \theta = -21 $ on the cost function below.



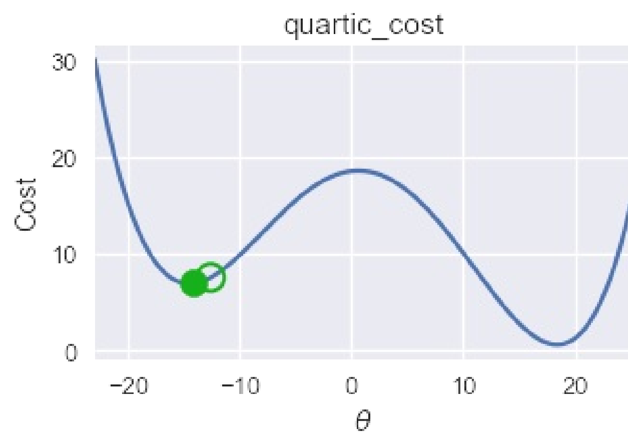

```
old theta: -21
new theta: -9.944999999999999
```



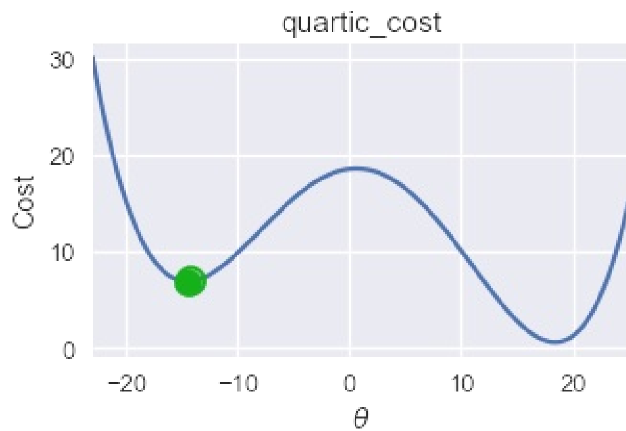

```
old theta: -9.9
new theta: -12.641412
```




```
old theta: -12.6
new theta: -14.162808
```

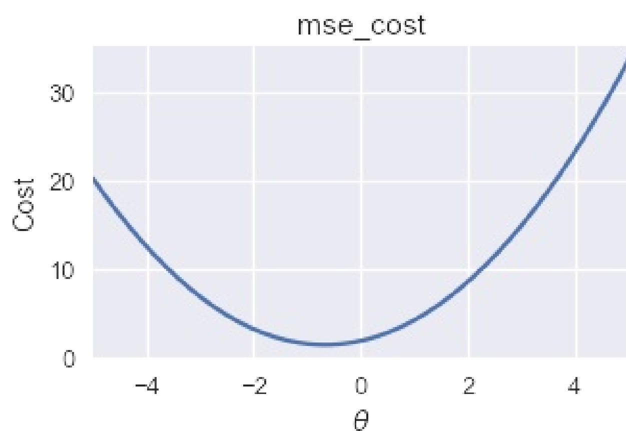



```
old theta: -14.2
new theta: -14.497463999999999
```



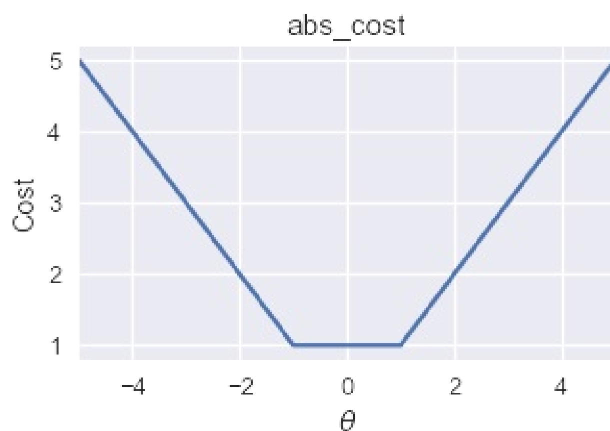
On this cost function and $ \theta $ value, gradient descent converges to $ \theta = -14.5 $, producing a cost of roughly 8. However, the global minimum for this cost function is $ \theta = 18 $, corresponding to a cost of nearly zero. From this example, we observe that gradient descent finds a *local minimum* which may not necessarily have the same cost as the *global minimum*.

Luckily, a number of useful cost functions have identical local and global minima. Consider the familiar mean squared error cost function, for example:



Running gradient descent on this cost function with an appropriate learning rate will always find the globally optimal $ \theta $ since the sole local minimum is also the global minimum.

The mean absolute error cost sometimes has multiple local minima. However, all the local minima produce the globally lowest cost possible.



On this cost function, gradient descent will converge to one of the local minima in the range $[-1, 1]$. Since all of these local minima have the lowest cost possible for this function, gradient descent will still return an optimal choice of θ.

Definition of Convexity¶

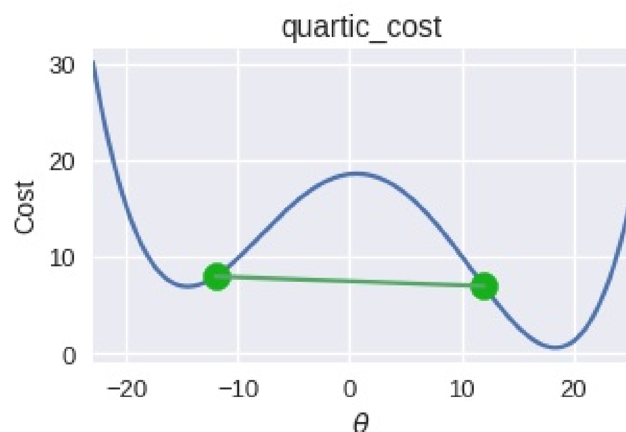
For some functions, any local minimum is also a global minimum. This set of functions are called **convex functions** since they curve upward. For a constant model, the MSE, mean absolute error, and Huber cost are all convex.

With an appropriate learning rate, gradient descent finds the globally optimal θ for convex cost functions. Because of this useful property, we prefer to fit our models using convex cost functions unless we have a good reason not to.

Formally, a function f is convex if and only if it satisfies the following inequality for all possible function inputs a and b, for all $t \in [0, 1]$:

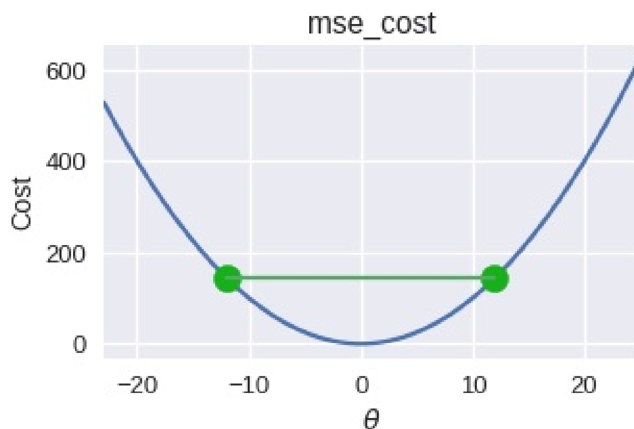
$$tf(a) + (1-t)f(b) \geq f(ta + (1-t)b)$$

This inequality states that all lines connecting two points of the function must reside on or above the function itself. For the cost function at the start of the section, we can easily find such a line that appears below the graph:



Thus, this cost function is non-convex.

For the MSE cost, all lines connecting two points of the graph appear above the graph. We plot one such line below.



The mathematical definition of convexity gives us a precise way of determining whether a function is convex. In this textbook, we will omit mathematical proofs of convexity and will instead state whether a chosen cost function is convex.

Summary¶

For a convex function, any local minimum is also a global minimum. This useful property allows gradient descent to efficiently find the globally optimal model parameters for a given cost function. While gradient descent will converge to a local minimum for non-convex cost functions, these local minima are not guaranteed to be globally optimal.

Linear Models

Now that we have a general method for fitting a model to a cost function, we turn our attention to improvements on our model. For the sake of simplicity, we previously restricted ourselves to a constant model: our model only ever predicts a single number.

However, giving our waiter such a model would hardly satisfy him. He would likely point out that he collected much more information about his tables than simply the tip percents. Why didn't we use his other data—e.g. size of the table or total bill—in order to make our model more useful?

In this chapter we will introduce linear models which will allow us to make use of our entire dataset to make predictions. Linear models are not only widely used in practice but also have rich theoretical underpinnings that will allow us to understand future tools for modeling. We introduce a simple linear regression model that uses one explanatory variable, explain how gradient descent is used to fit the model, and finally extend the model to incorporate multiple explanatory variables.

Show Widgets Open on DataHub

Table of Contents

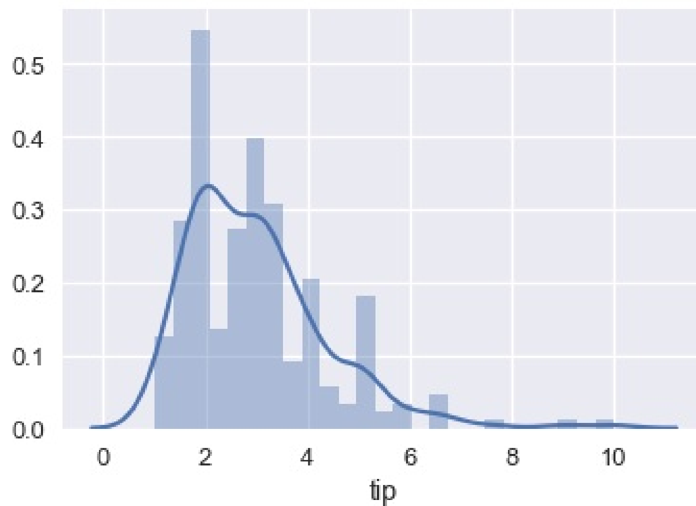
Predicting Tip Amounts¶

Previously, we worked with a dataset that contained one row for each table that a waiter served in a week. Our waiter collected this data in order to predict the tip amount he could expect to receive from a future table.

```
tips = sns.load_dataset('tips')
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

```
sns.distplot(tips['tip'], bins=25);
```



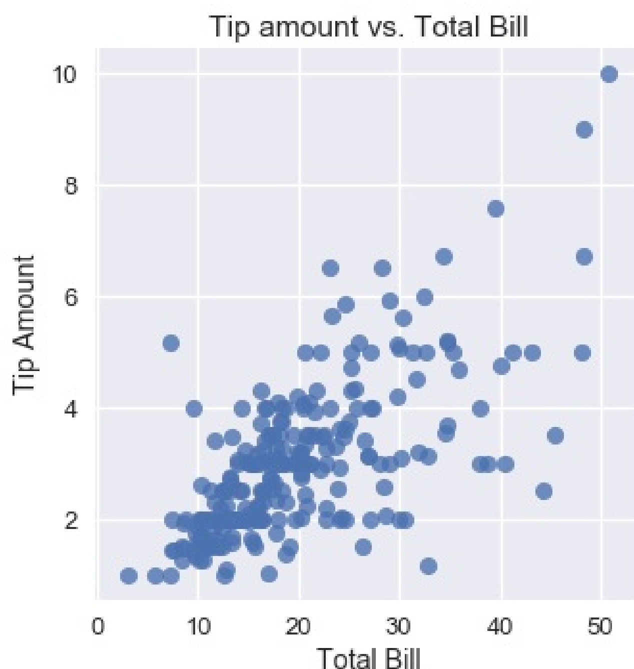
As we have covered previously, if we choose a constant model and the mean squared error cost, our model will predict the mean of the tip amount:

```
np.mean(tips['tip'])
```

```
2.9982786885245902
```

This means that if a new party orders a meal and the waiter asks us how much tip he will likely receive, we will say "around \$3", no matter how large the table is or how much their total bill was.

However, looking at other variables in the dataset suggest that we might be able to make more accurate predictions if we incorporate them into our model. For example, the following plot of the tip amount against the total bill shows a positive association.



Although the average tip amount is \$3, if a table orders \$40 worth of food we would certainly expect that the waiter receives more than \$3 of tip. Thus, we would like to alter our model so that it makes predictions based on the variables in our dataset instead of blindly predicting the mean tip amount. To do this, we use a linear model instead of constant one.

Let's briefly review our current toolbox for modeling and estimation and define some new notation so that we can better represent the additional complexity that linear models have.

Defining a Simple Linear Model¶

We are interested in predicting the tip amount based on the total bill of a table. Let $ y $ represent the tip amount, the variable we are trying to predict. Let $ x $ represent the total bill, the variable we use for prediction.

We define a linear model $ f_\theta $ that depends on $ x $:

$$ f_\theta (x) = \theta_1 x + \theta_0 $$

We treat $ f_\theta (x) $ as the underlying function that generated the data.

$ f_\theta (x) $ assumes that in truth, $ y $ has a perfectly linear relationship with $ x $. However, our observed data do not follow a perfectly straight line because of some random noise $ \epsilon $. Mathematically, we account for this by adding a noise term:

$$ y = f_\theta (x) + \epsilon $$

If the assumption that $ y $ has a perfectly linear relationship with $ x $ holds, and we are able to somehow find the exact values of $ \theta_1 $ and $ \theta_0 $, and we magically have no random noise, we will be able to perfectly predict the amount of tip the waiter will get for all tables, forever. Of course, we cannot completely fulfill any of these criteria in practice. Instead, we will estimate $ \theta_1 $ and $ \theta_0 $ using our dataset to make our predictions as accurate as possible.

Estimating the Linear Model¶

Since we cannot find $ \theta_1 $ and $ \theta_0 $ exactly, we will assume that our dataset approximates our population and use our dataset to estimate these parameters. We denote our estimations with $ \hat{\theta_1} $ and $ \hat{\theta_0} $ and define our model as:

$$ f_\hat{\theta} (x) = \hat{\theta_1} x + \hat{\theta_0} $$

Sometimes you will see $ h(x) $ written instead of $ f_\hat{\theta} (x) $; the "$ h $" stands for hypothesis, as $ f_\hat{\theta} (x) $ is our hypothesis of $ f_{\theta} (x) $.

In order to determine $ \hat{\theta_1} $ and $ \hat{\theta_0} $, we choose a cost function and minimize it using gradient descent.

Show Widgets Open on DataHub

Table of Contents

Fitting a Linear Model Using Gradient Descent¶

We want to fit a linear model that predicts the tip amount based on the total bill of the table:

$$ f_\hat{\theta} (x) = \hat{\theta_1} x + \hat{\theta_0} $$

In order to find $ \hat{\theta_1} $ and $ \hat{\theta_0} $, we need to first choose a cost function. We will choose the mean squared error cost function:

$$ \begin{aligned} L(\hat{\theta}, x, y) &= \frac{1}{n} \sum_{i = 1}^{n}(y_i - f_\hat{\theta}(x_i))^2\\ \end{aligned} $$

Note that we have modified our loss function to reflect the addition of an explanatory variable in our new model. Now, $ x $ is a vector containing the individual total bills, $ y $ is a vector containing the individual tip amounts, and $ \hat{\theta} $ is a vector: $ \theta = [\hat{\theta_1}, \hat{\theta_0}] $.

Using a linear model with the squared error also goes by the name of least-squares linear regression. We can use gradient descent to find the $ \theta $ that minimizes the cost.

An Aside on Using Correlation

If you have seen least-squares linear regression before, you may recognize that we can compute the correlation coefficient and use it to determine $ \hat{\theta_1} $ and $ \hat{\theta_0} $. This is simpler and faster to compute than using gradient descent for this specific problem, similar to how computing the mean was simpler than using gradient descent to fit a constant model. We will use gradient descent anyway because it is a general-purpose method of cost minimization that still works when we later introduce models that do not have analytic solutions. In fact, in many real-world scenarios we will use gradient descent even when an analytic solution exists because computing the analytic solution can take longer than gradient descent, especially on large datasets.

Derivative of the MSE cost¶

In order to use gradient descent, we have to compute the derivative of the MSE cost with respect to $ \hat{\theta} $. Now that $ \hat{\theta} $ is a vector of length 2 instead of a scalar, $ \nabla_{\hat\theta} L(\hat{\theta}, x, y) $ will also be a vector of length 2.

$$ \begin{aligned} \nabla_{\hat\theta} L(\hat{\theta}, x, y) &= \nabla_{\hat\theta} \left[\frac{1}{n} \sum_{i = 1}^{n}(y_i - f_\hat{\theta} (x_i))^2 \right] \\ &= \frac{1}{n} \sum_{i = 1}^{n}2 (y_i - f_\hat{\theta} (x_i))(- \nabla_{\hat\theta} f_\hat{\theta} (x_i))\\ &= -\frac{2}{n} \sum_{i = 1}^{n} (y_i - f_\hat{\theta} (x_i))(\nabla_{\hat\theta} f_\hat{\theta} (x_i))\\ \end{aligned} $$

We know:

$$ f_\hat{\theta} (x) = \hat{\theta_1} x + \hat{\theta_0} $$

We now need to compute $ \nabla_{\hat\theta} f_\hat{\theta} (x_i) $ which is a length 2 vector.

$$ \begin{aligned} \nabla_{\hat\theta} f_\hat{\theta} (x_i) &= \begin{bmatrix} \frac{\partial}{\partial \theta_0} f_\hat{\theta} (x_i)\\ \frac{\partial}{\partial \theta_1} f_\hat{\theta} (x_i) \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial}{\partial \theta_0} [\hat{\theta_1} x_i + \hat{\theta_0}]\\ \frac{\partial}{\partial \theta_1} [\hat{\theta_1} x_i + \hat{\theta_0}] \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ x_i \end{bmatrix} \\ \end{aligned} $$

Finally, we plug back into our formula above to get

$$ \begin{aligned} \nabla_{\hat\theta} L(\hat{\theta}, x, y) &= -\frac{2}{n} \sum_{i = 1}^{n}(y_i - f_\hat{\theta} (x_i))(\nabla_{\hat\theta} f_\hat{\theta} (x_i))\\ &= -\frac{2}{n} \sum_{i = 1}^{n} (y_i - f_\hat{\theta} (x_i)) \begin{bmatrix} 1 \\ x_i \end{bmatrix} \\ &= -\frac{2}{n} \sum_{i = 1}^{n} \begin{bmatrix} (y_i - f_\hat{\theta} (x_i)) \\ (y_i - f_\hat{\theta} (x_i)) x_i \end{bmatrix} \\ \end{aligned} $$

This is a length 2 vector since $ (y_i - f_\hat{\theta} (x_i)) $ is a scalar.

Running Gradient Descent¶

Now, let's fit a linear model on the tips dataset to predict the tip amount from the total table bill.

First, we define a Python function to compute the cost:

```
def simple_linear_model(thetas, x_vals):
    '''Returns predictions by a linear model on x_vals.'''
    return thetas[0] + thetas[1] * x_vals

def mse_cost(thetas, x_vals, y_vals):
    return np.mean((y_vals - simple_linear_model(thetas,
x_vals)) ** 2)
```

Then, we define a function to compute the gradient of the cost:

```
def grad_mse_cost(thetas, x_vals, y_vals):
    n = len(x_vals)
    grad_0 = y_vals - simple_linear_model(thetas, x_vals)
    grad_1 = (y_vals - simple_linear_model(thetas, x_vals)) *
x_vals
    return -2 / n * np.array([np.sum(grad_0), np.sum(grad_1)])
```

We'll use the previously defined `minimize` function that runs gradient descent, accounting for our new explanatory variable. It has the function signature (body omitted):

```
minimize(cost_fn, grad_cost_fn, x_vals, y_vals)
```

Finally, we run gradient descent!

```
%%time

thetas = minimize(mse_cost, grad_mse_cost, tips['total_bill'],
tips['tip'])
```

```
theta: [0. 0.] | cost: 10.896283606557377
theta: [0.   0.07] | cost: 3.8937622006094705
theta: [0.  0.1] | cost: 1.9359443267168215
theta: [0.01 0.12] | cost: 1.388538448286097
theta: [0.01 0.13] | cost: 1.235459416905535
theta: [0.01 0.14] | cost: 1.1926273731479433
theta: [0.01 0.14] | cost: 1.1806184944517062
theta: [0.01 0.14] | cost: 1.177227251696266
theta: [0.01 0.14] | cost: 1.1762453624313751
theta: [0.01 0.14] | cost: 1.1759370980989148
theta: [0.01 0.14] | cost: 1.175817178966766
CPU times: user 272 ms, sys: 67.3 ms, total: 339 ms
Wall time: 792 ms
```

We can see that gradient descent converges to the theta values of $\theta_0 = 0.01$ and $\theta_1 = 0.14$. Our linear model is:

$y = 0.14x + 0.01$

We can use our estimated thetas to make and plot our predictions alongside the original data points.



We can see that if a table's bill is \$10, our model will predict that the waiter gets around \$1.50 in tip. Similarly, if a table's bill is \$40, our model will predict a tip of around \$6.00.

Show Widgets Open on DataHub

Table of Contents

Multiple Linear Regression¶

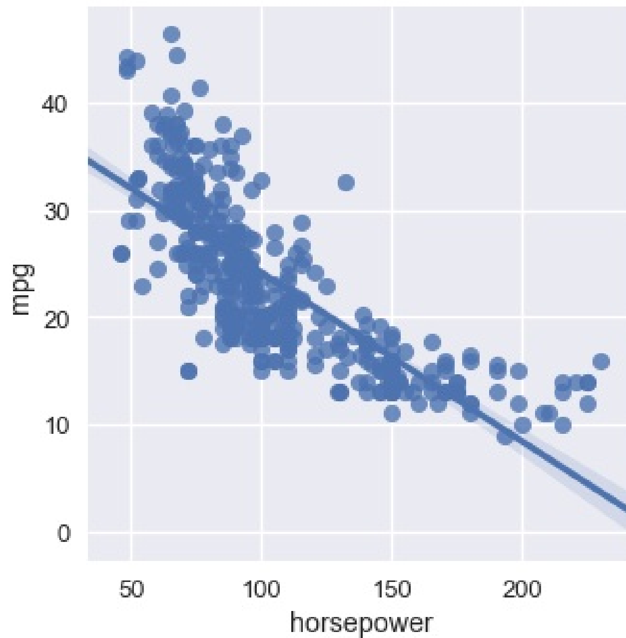
Our simple linear model has a key advantage over the constant model: it uses the data when making predictions. However, it is still rather limited since simple linear models only use one variable in our dataset. Many datasets have many potentially useful variables, and multiple linear regression can take advantage of that. For example, consider the following dataset on car models and their milage per gallon (MPG):

```
mpg = pd.read_csv('mpg.csv').dropna().reset_index(drop=True)
mpg
```

	mpg	cylinders	displacement	...	model year	origin	car name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
...	...	...	...	...	...	...	...
389	32.0	4	135.0	...	82	1	dodge rampage
390	28.0	4	120.0	...	82	1	ford ranger
391	31.0	4	119.0	...	82	1	chevy s-10

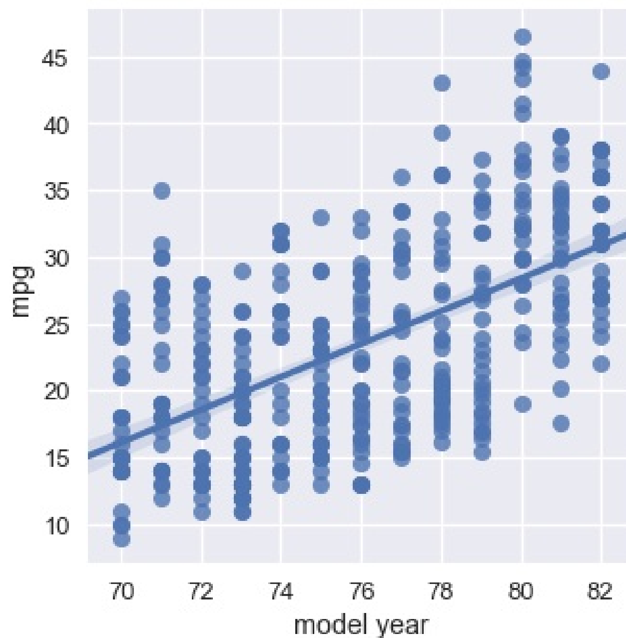
392 rows × 9 columns

It seems likely that multiple attributes of a car model will affect its MPG. For example, the MPG seems to decrease as horsepower increases:



However, cars released later generally have better MPG than older cars:

```
sns.lmplot(x='model year', y='mpg', data=mpg);
```



It seems possible that we can get a more accurate model if we could take both horsepower and model year into account when making predictions about the MPG. In fact, perhaps the best model takes into account all the numerical variables in our dataset. We can extend our univariate linear regression to allow prediction based on any number of attributes.

We state the following model:

$$ f_\hat{\theta} (x) = \hat{\theta_0} + \hat{\theta_1} x_1 + \ldots + \hat{\theta_p} x_p $$

Where $ x $ now represents a vector containing $ p $ attributes of a single car. The model above says, "Take multiple attributes of a car, multiply them by some weights, and add them together to make a prediction for MPG."

For example, if we're making a prediction on the first car in our dataset using the horsepower, weight, and model year columns, the vector $ x $ looks like:

	horsepower	weight	model year
0	130.0	3504.0	70

In there examples, we've kept the column names for clarity but keep in mind that $ x $ only contains the numerical values of the table above: $ x = [130.0, 3504.0, 70] $.

Now, we will perform a notational trick that will greatly simplify later formulas. We will prepend the value $ 1 $ to the vector $ x $, so that we have the following vector for $ x $:

	bias	horsepower	weight	model year
0	1	130.0	3504.0	70

Now, observe what happens to the formula for our model:

$$ \begin{aligned} f_\hat{\theta} (x) &= \hat{\theta_0} + \hat{\theta_1} x_1 + \ldots + \hat{\theta_p} x_p \\ &= \hat{\theta_0} (1) + \hat{\theta_1} x_1 + \ldots + \hat{\theta_p} x_p \\ &= \hat{\theta_0} x_0 + \hat{\theta_1} x_1 + \ldots + \hat{\theta_p} x_p \\ f_\hat{\theta} (x) &= \hat{\theta} \cdot x \end{aligned} $$

Where $ \hat{\theta} \cdot x $ is the vector dot product of $ \hat{\theta} $ and $ x $. Vector and matrix notation was designed to succinctly write linear combinations and is thus well-suited for our linear models. However, you will have to remember from now on that $ \hat{\theta} \cdot x $ is a vector-vector dot product. When in doubt, you can always expand the dot product into simple multiplications and additions.

Now, we define the matrix $ X $ as the matrix containing every car model as a row and a first column of biases. For example, here are the first five rows of $ X $:

	bias	horsepower	weight	model year
0	1	130.0	3504.0	70
1	1	165.0	3693.0	70
2	1	150.0	3436.0	70
3	1	150.0	3433.0	70
4	1	140.0	3449.0	70

Again, keep in mind that the actual matrix $ X $ only contains the numerical values of the table above.

Notice that $ X $ is composed of multiple $ x $ vectors stacked on top of each other. To keep the notation clear, we define $ X_{i} $ to refer to the row with index $ i $ of $ X $. We define $ X_{i,j} $ to refer to the element with index $ j $ of the row with index $ i $ of $ X $. Thus, $ X_i $ is a $ p $-dimensional vector and $ X_{i,j} $ is a scalar. $ X $ is an $ n \times p $ matrix, where n is the number of car examples we have and p is the number of attributes we have for a single car.

For example, from the table above we have $ X_4 = [1, 140, 3449, 70] $ and $ X_{4,1} = 140 $. This notation becomes important when we define the cost function since we will need both $ X $, the matrix of input values, and $ y $, the vector of MPGs.

MSE Cost and its Gradient¶

The mean squared error cost function takes in a vector of weights $ \hat{\theta} $, a matrix of inputs $ X $, and a vector of observed MPGs $ y $:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y_i - f_\hat{\theta} (X_i))^2\\ \end{aligned} $$

We've previously derived the gradient of the MSE cost with respect to $ \hat{\theta} $:

$$ \begin{aligned} \nabla_{\hat\theta} L(\hat{\theta}, X, y) &= -\frac{2}{n} \sum_{i}(y_i - f_\hat{\theta} (X_i))(\nabla_{\hat\theta} f_\hat{\theta} (X_i))\\ \end{aligned} $$

We know that:

$$ \begin{aligned} f_\hat{\theta} (x) &= \hat{\theta} \cdot x \\ \end{aligned} $$

Let's now compute $ \nabla_{\hat\theta} f_\hat{\theta} (x) $. The result is surprisingly simple because $ \hat{\theta} \cdot x = \hat{\theta_0} x_0 + \ldots + \hat{\theta_p} x_p $ and thus $ \frac{\partial}{\partial \theta_0}(\hat{\theta} \cdot x) = x_0 $, $ \frac{\partial}{\partial \theta_1}(\hat{\theta} \cdot x) = x_1 $, and so on.

$$ \begin{aligned} \nabla_{\hat\theta} f_\hat{\theta} (x) &= \nabla_{\hat\theta} [\hat{\theta} \cdot x] \\ &= \begin{bmatrix} \frac{\partial}{\partial \theta_0} (\hat{\theta} \cdot x) \\ \frac{\partial}{\partial \theta_1} (\hat{\theta} \cdot x) \\ \vdots \\ \frac{\partial}{\partial \theta_p} (\hat{\theta} \cdot x) \\ \end{bmatrix} \\ &= \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_p \end{bmatrix} \\ \nabla_{\hat\theta} f_\hat{\theta} (x) &= x \end{aligned} $$

Finally, we plug this result back into our gradient calculations:

$$ \begin{aligned} \nabla_{\hat\theta} L(\hat{\theta}, X, y) &= -\frac{2}{n} \sum_{i}(y_i - f_\hat{\theta} (X_i))(\nabla_{\hat\theta} f_\hat{\theta} (X_i))\\ &= -\frac{2}{n} \sum_{i}(y_i - \hat{\theta} \cdot X_i)(X_i)\\ \end{aligned} $$

Remember that since $ y_i - \hat{\theta} \cdot X_i $ is a scalar and $ X_i $ is a $ p $-dimensional vector, the gradient $ \nabla_{\hat\theta} L(\hat{\theta}, X, y) $ is a $ p $-dimensional vector.

We saw this same type of result when we computed the gradient for univariate linear regression and found that it was 2-dimensional since $ \theta $ was 2-dimensional.

Fitting the Model With Gradient Descent¶

We can now plug in our cost and its derivative into gradient descent. As usual, we will define the model, cost, and gradient cost in Python.

```
def linear_model(thetas, X):
    '''Returns predictions by a linear model on x_vals.'''
    return X @ thetas

def mse_cost(thetas, X, y):
    return np.mean((y - linear_model(thetas, X)) ** 2)

def grad_mse_cost(thetas, X, y):
    n = len(X)
    return -2 / n * (X.T @ y  - X.T @ X @ thetas)
```

Now, we can simply plug in our functions into our gradient descent minimizer:

```
%%time

thetas = minimize(mse_cost, grad_mse_cost, X, y)
print(f'theta: {thetas} | cost: {mse_cost(thetas, X, y):.2f}')
```

```
theta: [ 0.   0.   0.   0.] | cost: 610.47
theta: [ 0.    0.    0.01  0.  ] | cost: 178.95
theta: [ 0.01 -0.11 -0.    0.55] | cost: 15.78
theta: [ 0.01 -0.01 -0.01  0.58] | cost: 11.97
theta: [-4.   -0.01 -0.01  0.63] | cost: 11.81
theta: [-13.72  -0.    -0.01   0.75] | cost: 11.65
theta: [-13.72  -0.    -0.01   0.75] | cost: 11.65
CPU times: user 8.81 ms, sys: 3.11 ms, total: 11.9 ms
Wall time: 9.22 ms
```

According to gradient descent, our linear model is:

$y = -13.72 - 0.01x_2 + 0.75x_3$

Visualizing our Predictions¶

How does our model do? We can see that the cost decreased dramatically (from 610 to 11.6). We can show the predictions of our model alongside the original values:

	predicted_mpg	mpg	horsepower	weight	model year
0	15.447125	18.0	130.0	3504.0	70
1	14.053509	15.0	165.0	3693.0	70
2	15.785576	18.0	150.0	3436.0	70
...	...	...	...	...	...
389	32.456900	32.0	84.0	2295.0	82
390	30.354143	28.0	79.0	2625.0	82
391	29.726608	31.0	82.0	2720.0	82

392 rows × 5 columns

Since we found $ \hat{\theta} $ from gradient descent, we can verify for the first row of our data that $ \hat{\theta} \cdot X_0 $ matches our prediction above:

```
print(f'Prediction for first row: '
      f'{thetas[0] + thetas[1] * 130 + thetas[2] * 3504 +
thetas[3] * 70:.2f}')
```

```
Prediction for first row: 15.45
```

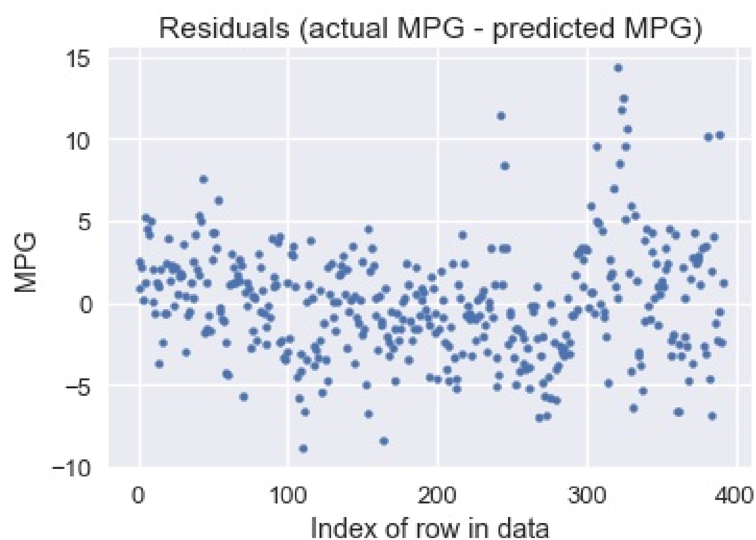
We've included a widget below to pan through the predictions and the data used to make the prediction:

Show Widget

```
(392 rows, 5 columns) total
```

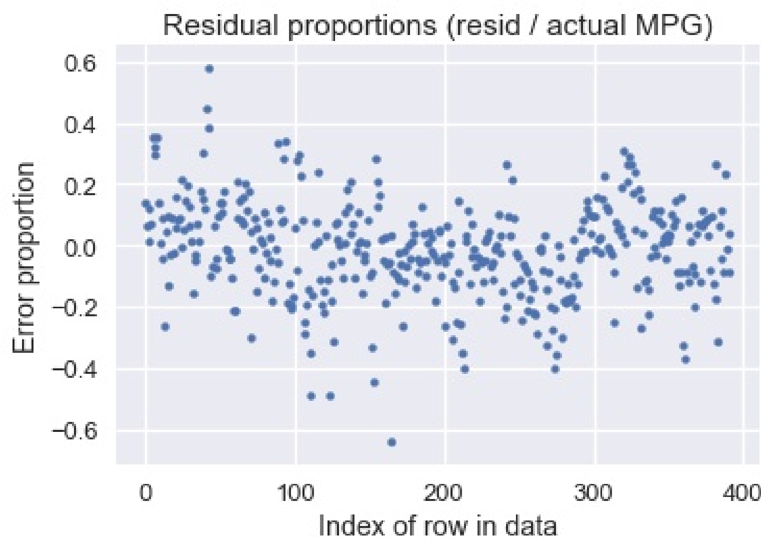
We can also plot the residuals of our predictions (actual values - predicted values):

```
resid = y - linear_model(thetas, X)
plt.scatter(np.arange(len(resid)), resid, s=15)
plt.title('Residuals (actual MPG - predicted MPG)')
plt.xlabel('Index of row in data')
plt.ylabel('MPG');
```



It looks like our model makes reasonable predictions for many car models, although there are some predictions that were off by over 10 MPG (some cars had under 10 MPG!). Perhaps we are more interested in the percent error between the predicted MPG values and the actual MPG values:

```
resid_prop = resid / with_predictions['mpg']
plt.scatter(np.arange(len(resid_prop)), resid_prop, s=15)
plt.title('Residual proportions (resid / actual MPG)')
plt.xlabel('Index of row in data')
plt.ylabel('Error proportion');
```



It looks like our model's predictions are usually within 20% away from the actual MPG values.

Using All the Data¶

Notice that in our example thus far, our $ X $ matrix has four columns: one column of all ones, the horsepower, the weight, and the model year. However, model allows us to handle an arbitrary number of columns:

$$ \begin{aligned} f_\hat{\theta} (x) &= \hat{\theta} \cdot x \end{aligned} $$

As we include more columns into our data matrix, we extend $ \theta $ so that it has one parameter for each column in $ X $. Instead of only selecting three numerical columns for prediction, why not use all seven of them?

	bias	cylinders	displacement	...	acceleration	model year	origin
0	1	8	307.0	...	12.0	70	1
1	1	8	350.0	...	11.5	70	1
2	1	8	318.0	...	11.0	70	1
...	...	...	...	...	...	...	...
389	1	4	135.0	...	11.6	82	1
390	1	4	120.0	...	18.6	82	1
391	1	4	119.0	...	19.4	82	1

392 rows × 8 columns

```
%%time

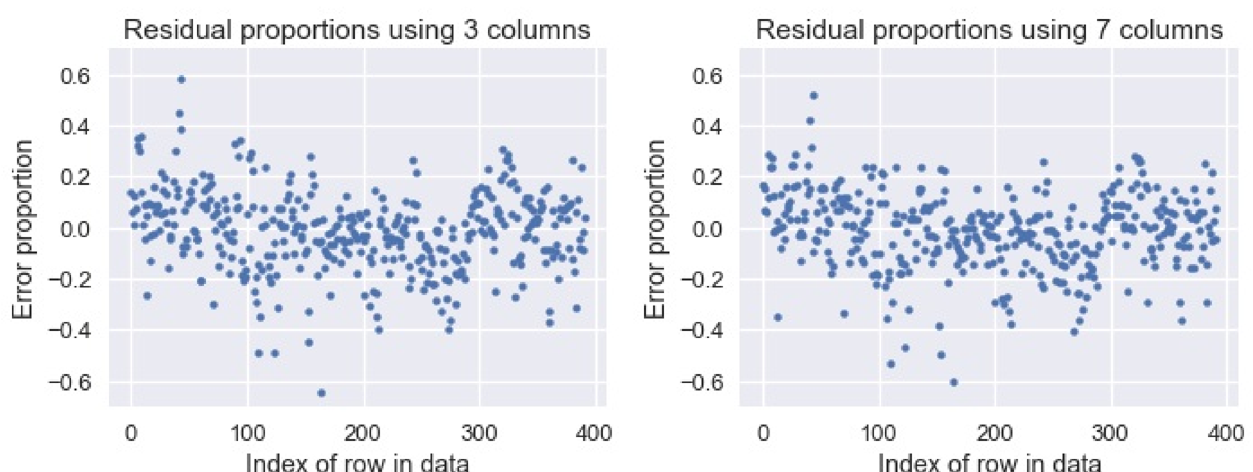
thetas_all = minimize(mse_cost, grad_mse_cost, X, y,
progress=10)
print(f'theta: {thetas_all} | cost: {mse_cost(thetas_all, X,
y):.2f}')
```

```
theta: [ 0.   0.   0.   0.   0.   0.   0.   0.] | cost: 610.47
theta: [-0.5  -0.81  0.02 -0.04 -0.01 -0.07  0.59  1.3 ] | cost:
11.22
theta: [-17.23  -0.49   0.02  -0.02  -0.01   0.08   0.75   1.43]
| cost: 10.85
theta: [-17.22  -0.49   0.02  -0.02  -0.01   0.08   0.75   1.43]
| cost: 10.85
CPU times: user 10.9 ms, sys: 3.51 ms, total: 14.4 ms
Wall time: 11.7 ms
```

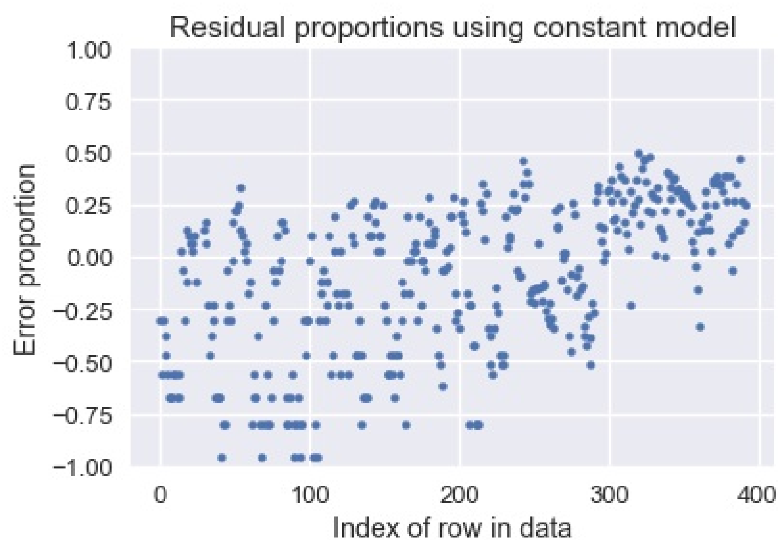
According to gradient descent, our linear model is:

$y = -17.22 - 0.49x_1 + 0.02x_2 - 0.02x_3 - 0.01x_4 + 0.08x_5 + 0.75x_6 + 1.43x_7$

We see that our cost has decreased from 11.6 with three columns of our dataset to 10.85 when using all seven numerical columns of our dataset. We display the proportion error plots for both old and new predictions below:



Although the difference is slight, you can see that the errors are a bit lower when using seven columns compared to using three. Both models are much better than using a constant model, as the below plot shows:



Using a constant model results in over 75% error for many car MPGs!

Summary¶

We have introduced the linear model for regression. Unlike the constant model, the linear regression model takes in features of our data into account when making predictions, making it much more useful whenever we have correlations between variables of our data.

The procedure of fitting a model to data should now be quite familiar:

1. Select a model.
2. Select a cost function.
3. Minimize the cost function using gradient descent.

It is useful to know that we can usually tweak one of the components without changing the others. In this section, we introduced the linear model without changing our cost function or using a different minimization algorithm. Although modeling can get complicated, it is usually easier to learn by focusing on one component at a time, then combining different parts together as needed in practice.

Feature Engineering

Feature engineering refers to the practice of creating and adding new features to the dataset itself in order to add complexity to our models.

So far we have only conducted linear regression using numerical features as the input—we used the (numeric) total bill in order to predict the tip amount. However, the tip dataset also contained categorical data, such as the day of week and the meal type. Feature engineering allows us to convert categorical variables into numerical features for linear regression.

Feature engineering also allows us to use our linear regression model to conduct polynomial regression by creating new variables in our dataset.

Show Widgets Open on DataHub

Table of Contents

The Walmart dataset¶

In 2014, Walmart released some of its sales data as part of a competition to predict the weekly sales of its stores. We've taken a subset of their data and loaded it below.

```
walmart = pd.read_csv('walmart.csv')
walmart
```

	Date	Weekly_Sales	IsHoliday	Temperature	Fuel_Price	Unempl
0	2010-02-05	24924.50	No	42.31	2.572	8.106
1	2010-02-12	46039.49	Yes	38.51	2.548	8.106
2	2010-02-19	41595.55	No	39.93	2.514	8.106
...	...	...	...	...	...	...
140	2012-10-12	22764.01	No	62.99	3.601	6.573
141	2012-10-19	24185.27	No	67.97	3.594	6.573
142	2012-10-26	27390.81	No	69.16	3.506	6.573

143 rows × 7 columns

The data contains several interesting features, including whether a week contained a holiday (`IsHoliday`), the unemployment rate that week (`Unemployment`), and which special deals the store offered that week (`MarkDown`).

Our goal is to create a model that predicts the `Weekly_Sales` variable using the other variables in our data. Using a linear regression model we directly can use the `Temperature` , `Fuel_Price` , and `Unemployment` columns because they contain numerical data.

Fitting a Model Using Scikit-Learn¶

In previous sections we have seen how to take the gradient of the cost function and use gradient descent to fit a model. To do this, we had to define Python functions for our model, the cost function, the gradient of the cost function, and the gradient descent algorithm. While this was important to demonstrate how the concepts work, in this section we will instead use a machine learning library called `scikit-learn` which allows us to fit a model with less code.

For example, to fit a multiple linear regression model using the numerical columns in the Walmart dataset, we first create a two-dimensional NumPy array containing the variables used for prediction and a one-dimensional array containing the values we want to predict:

```
numerical_columns = ['Temperature', 'Fuel_Price',
'Unemployment']
X = walmart[numerical_columns].as_matrix()
X
```

```
array([[ 42.31,   2.57,   8.11],
       [ 38.51,   2.55,   8.11],
       [ 39.93,   2.51,   8.11],
       ...,
       [ 62.99,   3.6 ,   6.57],
       [ 67.97,   3.59,   6.57],
       [ 69.16,   3.51,   6.57]])
```

```
y = walmart['Weekly_Sales'].as_matrix()
y
```

```
array([ 24924.5 ,  46039.49,  41595.55, ...,  22764.01,
24185.27,
        27390.81])
```

Then, we import the `LinearRegression` class from `scikit-learn` (docs), instantiate it, and call the `fit` method using `X` to predict `y`.

Note that previously we had to manually add a column of all 1's to the `X` matrix in order to conduct linear regression with an intercept. This time, `scikit-learn` will take care of the intercept column behind the scenes, saving us some work.

```
from sklearn.linear_model import LinearRegression

simple_classifier = LinearRegression()
simple_classifier.fit(X, y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

We are done! When we called `.fit`, `scikit-learn` found the linear regression parameters that minimized the least squares cost function. We can see the parameters below:

```
simple_classifier.coef_, simple_classifier.intercept_
```

```
(array([ -332.22,  1626.63,  1356.87]), 29642.700510138635)
```

To calculate the mean squared cost, we can ask the classifier to make predictions for the input data `X` and compare the predictions with the actual values `y`.

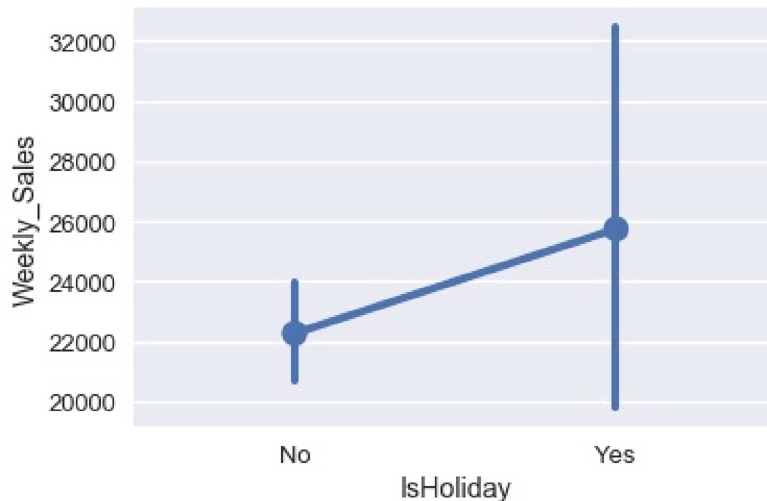
```
predictions = simple_classifier.predict(X)
np.mean((predictions - y) ** 2)
```

```
74401210.603607252
```

The mean squared error looks quite high. This is likely because our variables (temperature, price of fuel, and unemployment rate) are only weakly correlated with the weekly sales.

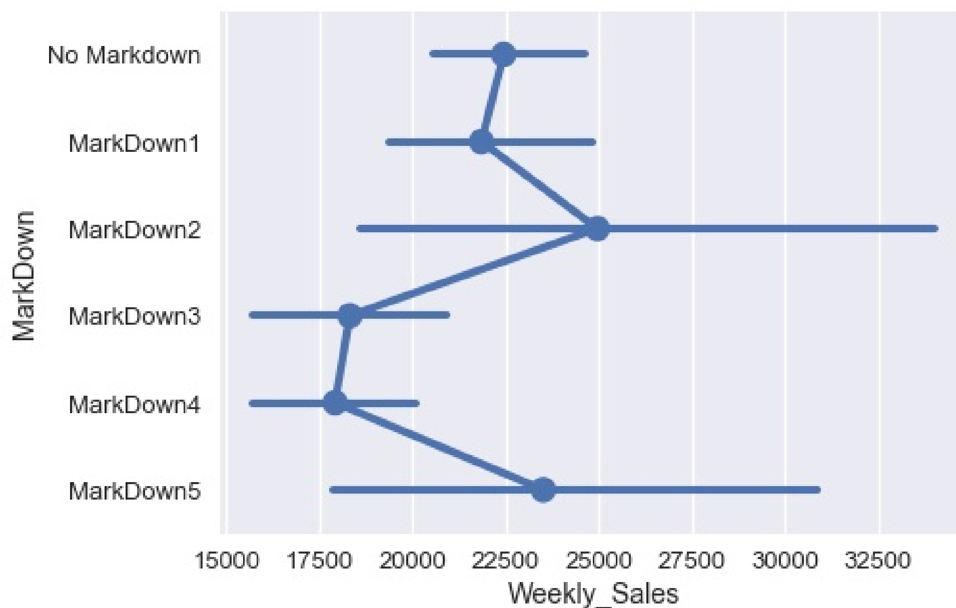
There are two more variables in our data that might be more useful for prediction: the `IsHoliday` column and `MarkDown` column. The boxplot below shows that holidays may have some relation with the weekly sales.

```
sns.pointplot(x='IsHoliday', y='Weekly_Sales', data=walmart);
```



The different markdown categories seem to correlate with different weekly sale amounts well.

```
markdowns = ['No Markdown', 'MarkDown1', 'MarkDown2',
'MarkDown3', 'MarkDown4', 'MarkDown5']
plt.figure(figsize=(7, 5))
sns.pointplot(x='Weekly_Sales', y='MarkDown', data=walmart,
order=markdowns);
```



However, both `IsHoliday` and `MarkDown` columns contain categorical data, not numerical, so we cannot use them as-is for regression.

The One-Hot Encoding¶

Fortunately, we can perform a **one-hot encoding** transformation on these categorical variables to convert them into numerical variables. The transformation works as follows: create a new column for every unique value in a categorical variable. The column contains a 1 if the variable originally had the corresponding value, otherwise the column contains a 0. For example, the `MarkDown` column below contains the following values:

	MarkDown
0	No Markdown
1	No Markdown
2	No Markdown
...	...
140	MarkDown2
141	MarkDown2
142	MarkDown1

143 rows × 1 columns

This variable contains six different unique values: 'No Markdown', 'MarkDown1', 'MarkDown2', 'MarkDown3', 'MarkDown4', and 'MarkDown5'. We create one column for each value to get six columns in total. Then, we fill in the columns with zeros and ones according the scheme described above.

	MarkDown=MarkDown1	MarkDown=MarkDown2	MarkDown=MarkDown3
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
...	...	...	...
140	0.0	1.0	0.0
141	0.0	1.0	0.0
142	1.0	0.0	0.0

143 rows × 6 columns

Notice that the first value in the data is "No Markdown", and thus only the last column of the first row in the transformed table is marked with 1. In addition, the last value in the data is "MarkDown1" which results in the first column of row 142 marked as 1.

Each row of the resulting table will contain a single column containing 1; the rest will contain 0. The name "one-hot" reflects the fact that only one column is "hot" (marked with a 1).

One-Hot Encoding in Scikit-Learn¶

To perform one-hot encoding we can use `scikit-learn`'s `DictVectorizer` class. To use the class, we have to convert our dataframe into a list of dictionaries. The DictVectorizer class automatically one-hot encodes the categorical data (which needs to be strings) and leaves numerical data untouched.

```
from sklearn.feature_extraction import DictVectorizer

all_columns = ['Temperature', 'Fuel_Price', 'Unemployment',
'IsHoliday',
               'MarkDown']

records = walmart[all_columns].to_dict(orient='records')
encoder = DictVectorizer(sparse=False)
encoded_X = encoder.fit_transform(records)
encoded_X
```

```
array([[  2.57,   1.  ,   0.  , ...,   1.  ,  42.31,   8.11],
       [  2.55,   0.  ,   1.  , ...,   1.  ,  38.51,   8.11],
       [  2.51,   1.  ,   0.  , ...,   1.  ,  39.93,   8.11],
       ...,
       [  3.6 ,   1.  ,   0.  , ...,   0.  ,  62.99,   6.57],
       [  3.59,   1.  ,   0.  , ...,   0.  ,  67.97,   6.57],
       [  3.51,   1.  ,   0.  , ...,   0.  ,  69.16,   6.57]])
```

To get a better sense of the transformed data, we can display it with the column names:

```
pd.DataFrame(data=encoded_X, columns=encoder.feature_names_)
```

	Fuel_Price	IsHoliday=No	IsHoliday=Yes	MarkDown=MarkDown1	..
0	2.572	1.0	0.0	0.0	..
1	2.548	0.0	1.0	0.0	..
2	2.514	1.0	0.0	0.0	..
...	...	...	...	...	..
140	3.601	1.0	0.0	0.0	..
141	3.594	1.0	0.0	0.0	..
142	3.506	1.0	0.0	1.0	..

143 rows × 11 columns

The numerical variables (fuel price, temperature, and unemployment) are left as numbers. The categorical variables (holidays and markdown) are one-hot encoded. When we use the new matrix of data to fit a linear regression model, we will generate one parameter for each column of the data. Since this data matrix contains eleven columns, the model will have twelve parameters since we fit extra parameter for the intercept term.

Fitting a Model Using the Transformed Data¶

We can now use the `encoded_X` variable for linear regression.

```
clf = LinearRegression()
clf.fit(encoded_X, y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

As promised, we have eleven parameters for the columns and one intercept parameter.

```
clf.coef_, clf.intercept_
```

```
(array([ 1622.11,    -2.04,     2.04,   962.91,  1805.06,
-1748.48,
        -2336.8 ,   215.06,  1102.25,  -330.91,  1205.56]),
29723.135729284979)
```

We can compare a few of the predictions from both classifiers to see whether there's a large difference between the two.

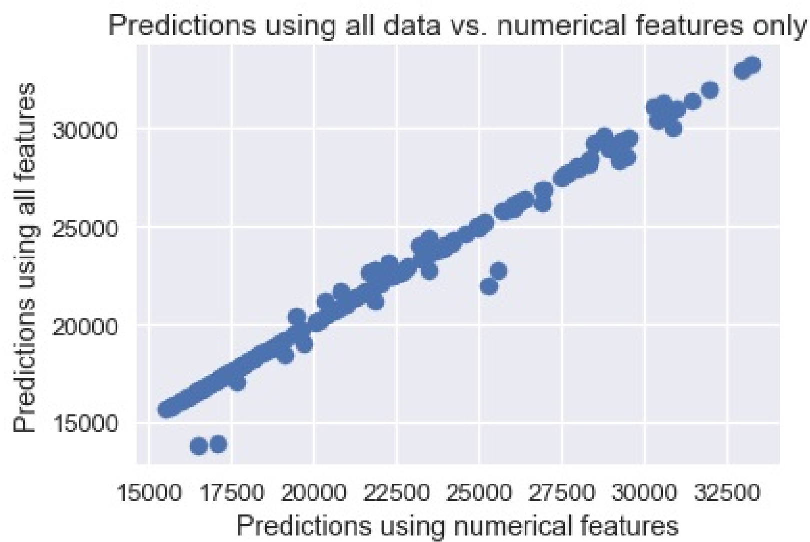
```
walmart[['Weekly_Sales']].assign(
    pred_numeric=simple_classifier.predict(X),
    pred_both=clf.predict(encoded_X)
)
```

	Weekly_Sales	pred_numeric	pred_both
0	24924.50	30768.878035	30766.790214
1	46039.49	31992.279504	31989.410395
2	41595.55	31465.220158	31460.280008
...	...	...	...
140	22764.01	23492.262649	24447.348979
141	24185.27	21826.414794	22788.049554
142	27390.81	21287.928537	21409.367463

143 rows × 3 columns

It appears that both models make very similar predictions. A scatter plot of both sets of predictions confirms this.

```
plt.scatter(simple_classifier.predict(X),
clf.predict(encoded_X))
plt.title('Predictions using all data vs. numerical features
only')
plt.xlabel('Predictions using numerical features')
plt.ylabel('Predictions using all features');
```



Model Diagnosis¶

Why might this be the case? We can examine the parameters that both models learn. The table below shows the weights learned by the classifier that only used numerical variables without one-hot encoding:

	0
Temperature	-332.221180
Fuel_Price	1626.625604
Unemployment	1356.868319
Intercept	29642.700510

The table below shows the weights learned by the classifier with one-hot encoding.

	0
Fuel_Price	1622.106239
IsHoliday=No	-2.041451
IsHoliday=Yes	2.041451
MarkDown=MarkDown1	962.908849
MarkDown=MarkDown2	1805.059613
MarkDown=MarkDown3	-1748.475046
MarkDown=MarkDown4	-2336.799791
MarkDown=MarkDown5	215.060616
MarkDown=No Markdown	1102.245760
Temperature	-330.912587
Unemployment	1205.564331
Intercept	29723.135729

We can see that even when we fit a linear regression model using one-hot encoded columns the weights for fuel price, temperature, and unemployment are very similar to the previous values. All the weights are small in comparison to the intercept term, suggesting that most of the variables are still only slightly correlated with the actual sale amounts. In fact, the model weights for the `IsHoliday` variable are so low that it makes nearly no difference in prediction whether the date was a holiday or not. Although some of the `MarkDown` weights are rather large, many markdown events only appear a few times in the dataset.

```
walmart['MarkDown'].value_counts()
```

```
No Markdown    92
MarkDown1      25
MarkDown2      13
MarkDown5       9
MarkDown4       2
MarkDown3       2
Name: MarkDown, dtype: int64
```

This suggests that we probably need to collect more data in order for the model to better utilize the effects of markdown events on the sale amounts. (In reality, the dataset shown here is a small subset of a much larger dataset released by Walmart. It will be a useful

exercise to train a model using the entire dataset instead of a small subset.)

Summary¶

We have learned to use one-hot encoding, a useful technique for conducting linear regression on categorical data. Although in this particular example the transformation didn't affect our model very much, in practice the technique is used widely when working with categorical data. One-hot encoding also illustrates the general principle of feature engineering—it takes an original data matrix and transforms it into a potentially more useful one.

Show Widgets Open on DataHub

Table of Contents

Predicting Ice Cream Ratings¶

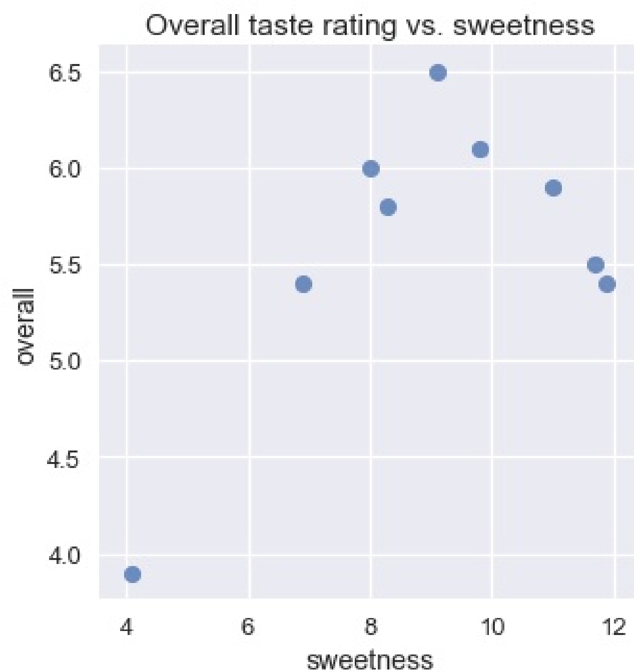
Suppose we are trying to create new, popular ice cream flavors. We are interested in the following regression problem: given the sweetness of an ice cream flavor, predict its overall taste rating out of 7.

```
ice = pd.read_csv('icecream.csv')
ice
```

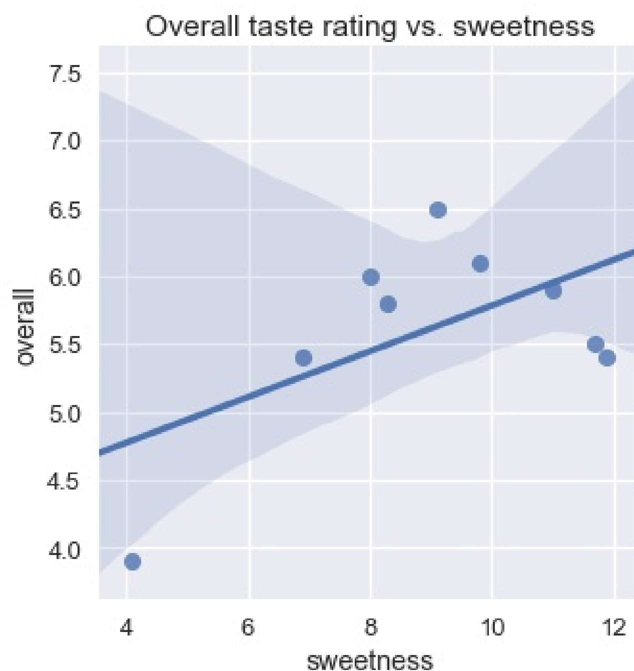
	sweetness	overall
0	4.1	3.9
1	6.9	5.4
2	8.3	5.8
...	...	...
6	11.0	5.9
7	11.7	5.5
8	11.9	5.4

9 rows × 2 columns

Although we expect that an ice cream flavor that is not sweet enough would receive a low rating, we also expect that an ice flavor that is too sweet would also receive a low rating. This is reflected in the scatter plot of overall rating and sweetness:



Unfortunately, a linear model alone cannot take this increase-then-decrease behavior into account; in a linear model, the overall rating can only increase or decrease monotonically with the sweetness. We can see that using linear regression results in a poor fit.



One useful approach for this problem is to fit a polynomial curve instead of line. Such a curve would allow us to model the fact that the overall rating increases with sweetness only up to a certain point, then decreases as sweetness increases.

With a feature engineering technique, we can simply add new columns to our data to use our linear model for polynomial regression.

Polynomial Features¶

Recall that in linear regression we fit one weight for each column of our data matrix $ X $. In this case, our matrix $ X $ contains two columns: a column of all ones and the sweetness.

	bias	sweetness
0	1.0	4.1
1	1.0	6.9
2	1.0	8.3
...	...	...
6	1.0	11.0
7	1.0	11.7
8	1.0	11.9

9 rows × 2 columns

Our model is thus:

$$ f_\hat{\theta} (x) = \hat{\theta_0} + \hat{\theta_1} \cdot \text{sweetness} $$

We can create a new column in $ X $ containing the squared values of the sweetness.

	bias	sweetness	sweetness^2
0	1.0	4.1	16.81
1	1.0	6.9	47.61
2	1.0	8.3	68.89
...	...	...	...
6	1.0	11.0	121.00
7	1.0	11.7	136.89
8	1.0	11.9	141.61

9 rows × 3 columns

Since our model learns one weight for each column of its input matrix, our model will become:

$$ f_\hat{\theta} (x) = \hat{\theta_0}

```
+ \hat{\theta_1} \cdot \text{sweetness}
+ \hat{\theta_2} \cdot \text{sweetness}^2
```

$$

Our model now fits a polynomial with degree two to our data. We can easily fit higher degree polynomials by adding columns for sweetness^3, sweetness^4, and so on.

Notice that this model is still a linear model because it is **linear in its parameters**—each $\hat{\theta_i}$ is a scalar value of degree one. However, the model is **polynomial in its features** because its input data contains a column that is a polynomial transformation of another column.

Polynomial Regression¶

To conduct polynomial regression, we use a linear model with polynomial features. Thus, we import the `LinearRegression` model and `PolynomialFeatures` transform from `scikit-learn`.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

Our original data matrix X contains the following values. Remember that we include the column and row labels for reference purposes only; the actual X matrix only contains the numerical data in the table below.

```
ice[['sweetness']]
```

	sweetness
0	4.1
1	6.9
2	8.3
...	...
6	11.0
7	11.7
8	11.9

9 rows × 1 columns

We first use the `PolynomialFeatures` class to transform the data, adding polynomial features of degree 2.

```
transformer = PolynomialFeatures(degree=2)
X = transformer.fit_transform(ice[['sweetness']])
X
```

```
array([[  1.  ,   4.1 ,  16.81],
       [  1.  ,   6.9 ,  47.61],
       [  1.  ,   8.3 ,  68.89],
       ...,
       [  1.  ,  11.  , 121.  ],
       [  1.  ,  11.7 , 136.89],
       [  1.  ,  11.9 , 141.61]])
```

Now, we fit a linear model to this data matrix.

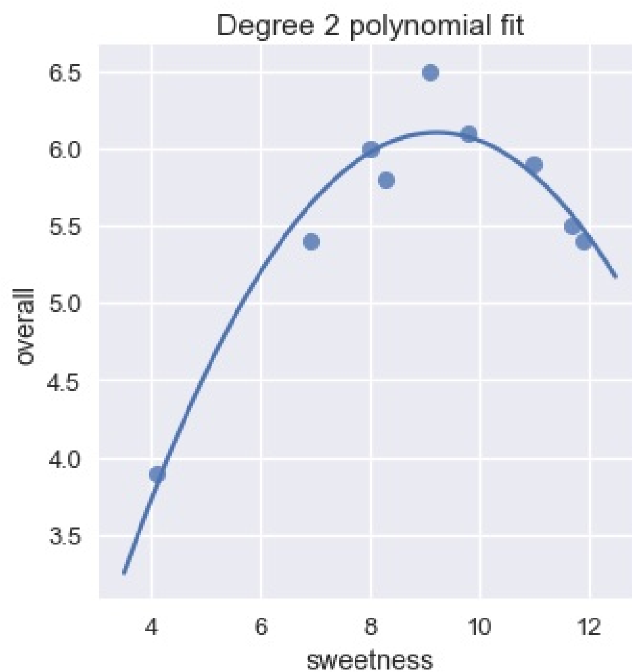
```
clf = LinearRegression(fit_intercept=False)
clf.fit(X, ice['overall'])
clf.coef_
```

```
array([-1.3 ,  1.6 , -0.09])
```

The parameters above show that for this dataset, the best-fit model is:

$$ f_\hat{\theta} (x) = -1.3 + 1.6 \cdot \text{sweetness} - 0.09 \cdot \text{sweetness}^2 $$

We can now compare this model's predictions against the original data.



This model looks like a much better fit than our linear model. We can also verify that the mean squared cost for the degree 2 polynomial fit is much lower than the cost for the linear fit.

```
MSE cost for linear reg:     0.323
MSE cost for deg 2 poly reg: 0.032
```

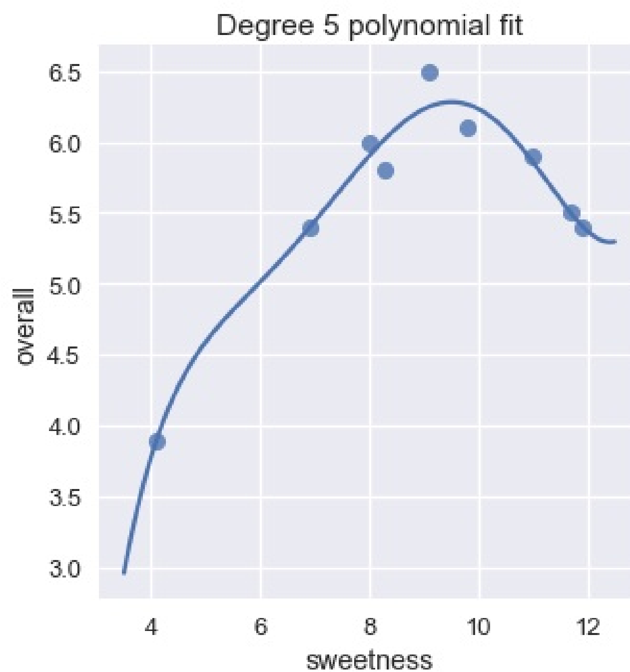
Increasing the Degree¶

As mentioned earlier, we are free to add higher degree polynomial features to our data. For example, we can easily create polynomial features of degree 5:

	bias	sweetness	sweetness^2	sweetness^3	sweetness^4	sweetn
0	1.0	4.1	16.81	68.921	282.5761	1158.56
1	1.0	6.9	47.61	328.509	2266.7121	15640.3
2	1.0	8.3	68.89	571.787	4745.8321	39390.4
...	...	...	...	...	...	...
6	1.0	11.0	121.00	1331.000	14641.0000	161051
7	1.0	11.7	136.89	1601.613	18738.8721	219244
8	1.0	11.9	141.61	1685.159	20053.3921	238635

9 rows × 6 columns

Fitting a linear model using these features results in a degree five polynomial regression.



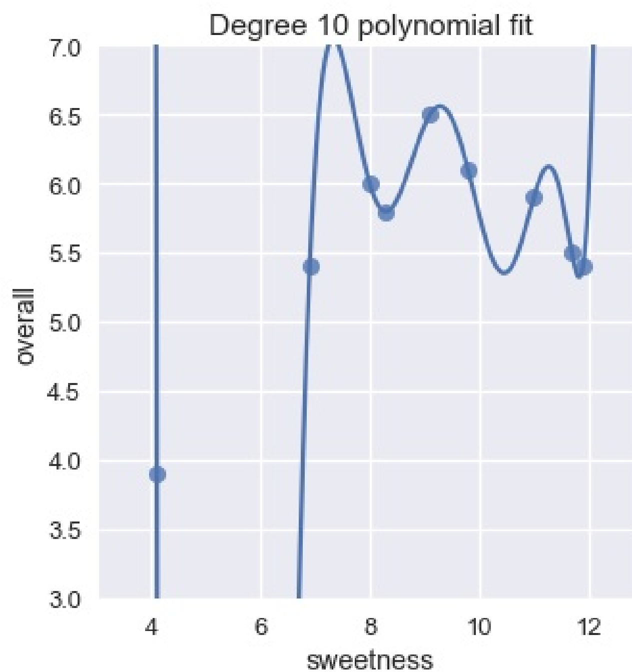
The plot shows that a degree five polynomial seems to fit the data roughly as well as a degree two polynomial. In fact, the mean squared cost for the degree five polynomial is almost half of the cost for the degree two polynomial.

```
pred_five = clf_five.predict(X_five)

print(f'MSE cost for linear reg:     {mse_cost(pred_linear,
y):.3f}')
print(f'MSE cost for deg 2 poly reg: {mse_cost(pred_quad,
y):.3f}')
print(f'MSE cost for deg 5 poly reg: {mse_cost(pred_five,
y):.3f}')
```

```
MSE cost for linear reg:     0.323
MSE cost for deg 2 poly reg: 0.032
MSE cost for deg 5 poly reg: 0.017
```

This suggests that we might do even better by increasing the degree even more. Why not a degree 10 polynomial?

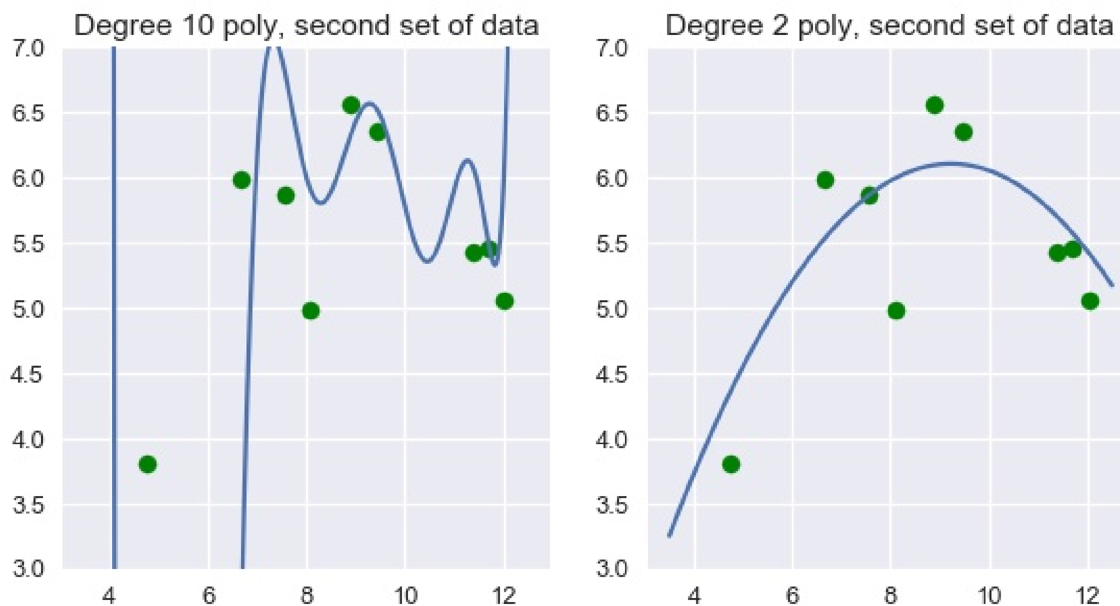


Here are the mean squared costs for the regression models we've seen thus far:

```
MSE cost for linear reg:      0.323
MSE cost for deg 2 poly reg:  0.032
MSE cost for deg 5 poly reg:  0.017
MSE cost for deg 10 poly reg: 0.000
```

The degree 10 polynomial has a cost of zero! This makes sense if we take a closer look at the plot; the degree ten polynomial manages to pass through the precise location of each point in the data.

However, you should feel hesitant to use the degree 10 polynomial to predict ice cream ratings. Intuitively, the degree 10 polynomial seems to fit our specific set of data too closely. If we take another set of data and plot them on the scatter plot above, we can expect that they fall close to our original set of data. When we do this, however, the degree 10 polynomial suddenly seems like a poor fit while the degree 2 polynomial still looks reasonable.



We can see that in this case, degree two polynomial features work better than both no transformation and degree ten polynomial features.

This raises the natural question: in general, how do we determine which degree polynomial to fit? Although we are tempted to use the cost on the training dataset to pick the best polynomial, we have seen that using this cost can pick a model that is too complex. Instead, we want to evaluate our model on data that is not used to fit parameters.

Summary¶

In this section, we introduce another feature engineering technique: adding polynomial features to the data in order to perform polynomial regression. Like one-hot encoding, adding polynomial features allows us to use our linear regression model effectively on more types of data.

We have also encountered a fundamental issue with feature engineering. Adding many features to the data gives the model a lower cost on its original set of data but often results in a less accurate model on new sets of data.

Show Widgets Open on DataHub

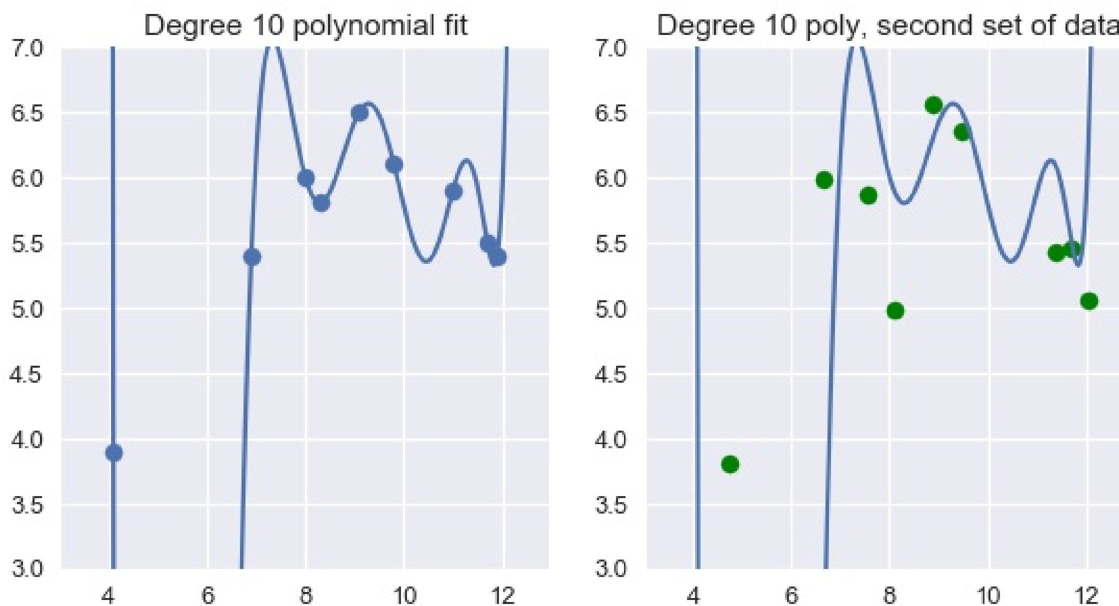
Table of Contents

Cross-Validation¶

In the previous section, we observe that the model cost on the training set, the dataset used to fit the model, can mislead. Adding more features to the data causes the training error to decrease. However, adding too many features causes overfitting—our model makes predictions based on patterns in the data generated by noise rather than the underlying phenomenon. To properly evaluate and select features, we turn to a technique called **cross-validation**.

Issues with Training Error¶

Using degree 10 polynomial features on the dataset below results in a perfectly accurate model for the training data. Unfortunately, this model fails to generalize to previously-unseen data from the population.



We cannot use the training error to pick features to add to the data since the training error will always favor more features. In order to accurately perform feature selection, we must check the model's error on data that is not used to fit the model.

Train-Validation-Test Split¶

To accomplish this, we randomly split the original dataset into three disjoint subsets:

- Training set: The data used to fit the model.
- Validation set: The data used to select features.
- Test set: The data used to report the model's final accuracy.

After splitting, we select a set of features and a model based on the following procedure:

1. For each potential set of features, fit a model using the training set. The error of a model on the training set is its *training error*.
2. Check the error of each model on the validation set: its *validation error*. Select the model that achieves the lowest validation error. This is the final choice of features and model.
3. Calculate the *test error*, error of the final model on the test set. This is the final reported accuracy of the model. We are forbidden from adjusting the features or model to decrease test error; doing so effectively converts the test set into a validation set. Instead, we must collect a new test set after making further changes to the features or the model.

The second step in the process above is called **cross-validation** (sometimes abbreviated as CV). Cross-validation allows us to more accurately determine the set of features to keep in the final model than using the training error alone.

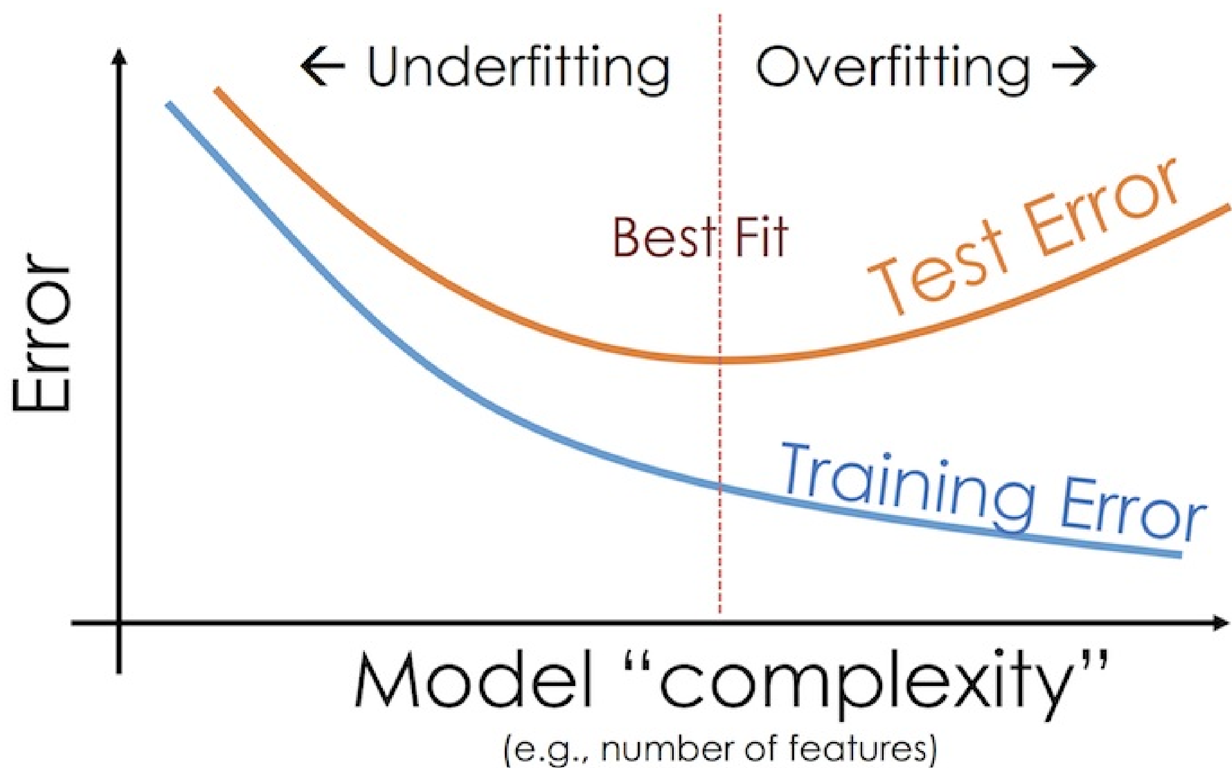
Size of the train-validation-test split

The train-validation-test split commonly uses 70% of the data as the training set, 15% as the validation set, and the remaining 15% as the test set. Increasing the size of the training set helps model accuracy but causes more variation in the validation and test error.

Training Error and Test Error¶

A model is of little use to us if it fails to generalize to unseen data from the population. The test error provides an accurate representation of the model's performance on new data since we do not use the test set to train the model or select features.

In general, the training error decreases as we add complexity to our model with additional features or more complex prediction mechanisms. The test error, on the other hand, decreases up to a certain amount of complexity then increases again as the model overfits the training set.



Feature Selection for Ice Cream Ratings¶

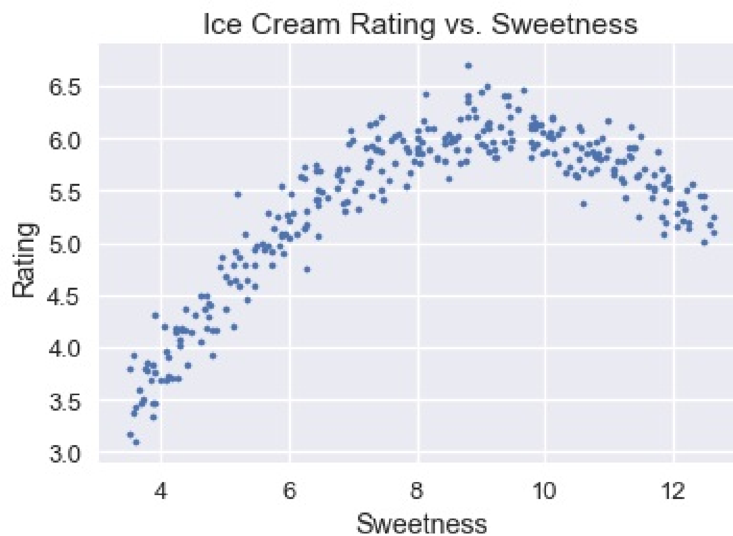
We use the complete model selection process, including cross-validation, to select a model that predicts ice cream ratings from ice cream sweetness. The complete ice cream dataset is shown below.

```
ice
```

	sweetness	overall
0	3.60	3.09
1	3.50	3.17
2	3.69	3.46
...	...	...
6	11.00	5.90
7	11.70	5.50
8	11.90	5.40

309 rows × 2 columns

We've included a scatter plot of the overall rating versus ice cream sweetness below.



We first partition our data into a training dataset, validation dataset, and test dataset using `scikit-learn`'s `sklearn.model_selection.train_test_split` method to perform a 70/15/15% train-validation-test split.

```
from sklearn.model_selection import train_test_split

valid_size, test_size = 46, 46

# To perform a three-way split, we need to call train_test_split
twice - once
# for the test set, one for the validation set.
X, X_test, y, y_test = train_test_split(
    ice[['sweetness']], ice['overall'], test_size=test_size)

X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=valid_size)

print(f'  Training set size: {len(X_train)}')
print(f'Validation set size: {len(X_valid)}')
print(f'      Test set size: {len(X_test)}')
```

```
  Training set size: 217
Validation set size: 46
      Test set size: 46
```

We now fit polynomial regression models using the training set `X_train` and `y_train`, one for each polynomial degree from 1 to 10.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# First, we add polynomial features to X_train
transformers = [PolynomialFeatures(degree=deg)
                for deg in range(1, 11)]
X_train_polys = [transformer.fit_transform(X_train)
                 for transformer in transformers]

# Display the X_train with degree 5 polynomial features
X_train_polys[4]
```

```
array([[     1.  ,      7.09,     50.27,    356.4 ,   2526.88,
17915.59],
       [     1.  ,      9.88,     97.61,    964.43,   9528.57,
94142.28],
       [     1.  ,     10.12,    102.41,   1036.43,  10488.71,
106145.74],
       ...,
       [     1.  ,     11.4 ,    129.96,   1481.54,  16889.6 ,
192541.46],
       [     1.  ,      5.2 ,     27.04,    140.61,    731.16,
3802.04],
       [     1.  ,      9.86,     97.22,    958.59,   9451.65,
93193.28]])
```

```
# Next, we train a linear regression classifier for each
featurized dataset.
# We set fit_intercept=False since the PolynomialFeatures
transformer adds the
# bias column for us.
clfs = [LinearRegression(fit_intercept=False).fit(X_train_poly,
y_train)
        for X_train_poly in X_train_polys]
```

After training the models, we evaluate the mean squared error of each model on the validation set.

```
def mse_cost(y_pred, y_actual):
    return np.mean((y_pred - y_actual) ** 2)

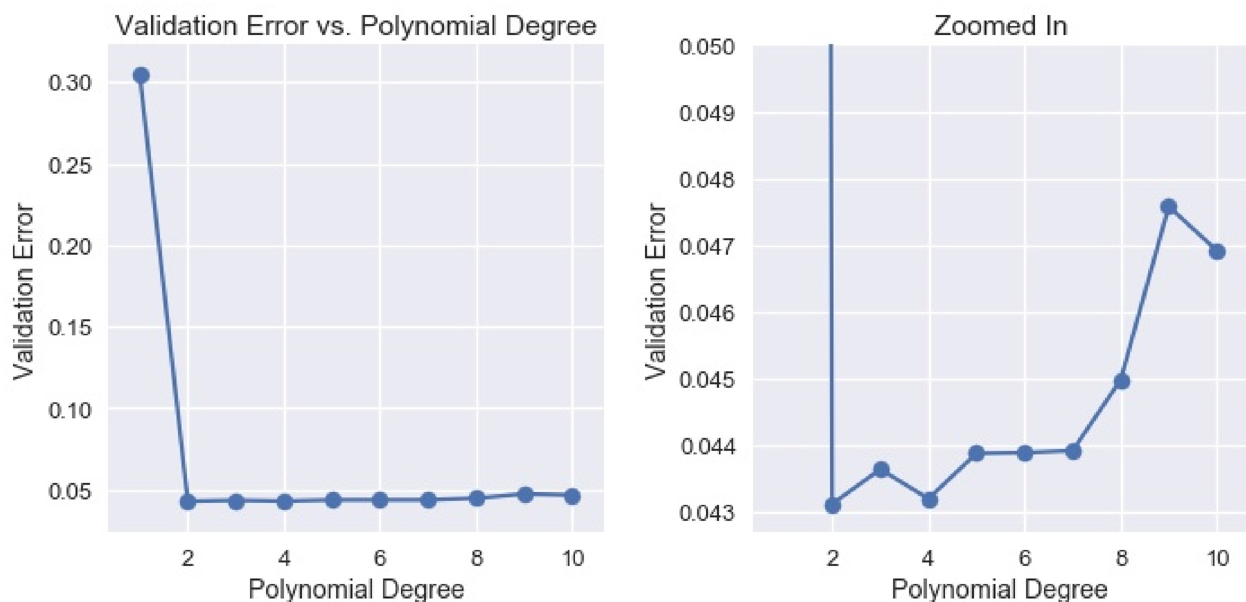
# To make predictions on the validation set we need to add
polynomial features
# to the validation set too.
X_valid_polys = [transformer.fit_transform(X_valid)
                 for transformer in transformers]

predictions = [clf.predict(X_valid_poly)
               for clf, X_valid_poly in zip(clfs,
X_valid_polys)]

costs = [mse_cost(pred, y_valid) for pred in predictions]
```

	Validation Error
Degree	
1	0.304539
2	0.043109
3	0.043629
4	0.043187
5	0.043868
6	0.043878
7	0.043909
8	0.044971
9	0.047573
10	0.046913

We can see that as we use higher degree polynomial features, the validation error decreases and increases again.



Examining the validation errors reveals the most accurate model only used degree 2 polynomial features. Thus, we select the degree 2 polynomial model as our final model and report its test error.

```
best_trans = transformers[1]
best_clf = clfs[1]

training_error = mse_cost(best_clf.predict(X_train_polys[1]),
y_train)
validation_error = mse_cost(best_clf.predict(X_valid_polys[1]),
y_valid)
test_error =
mse_cost(best_clf.predict(best_trans.transform(X_test)), y_test)

print('Degree 2 polynomial')
print(f'  Training error: {training_error:0.3f}')
print(f'Validation error: {validation_error:0.3f}')
print(f'      Test error: {test_error:0.3f}')
```

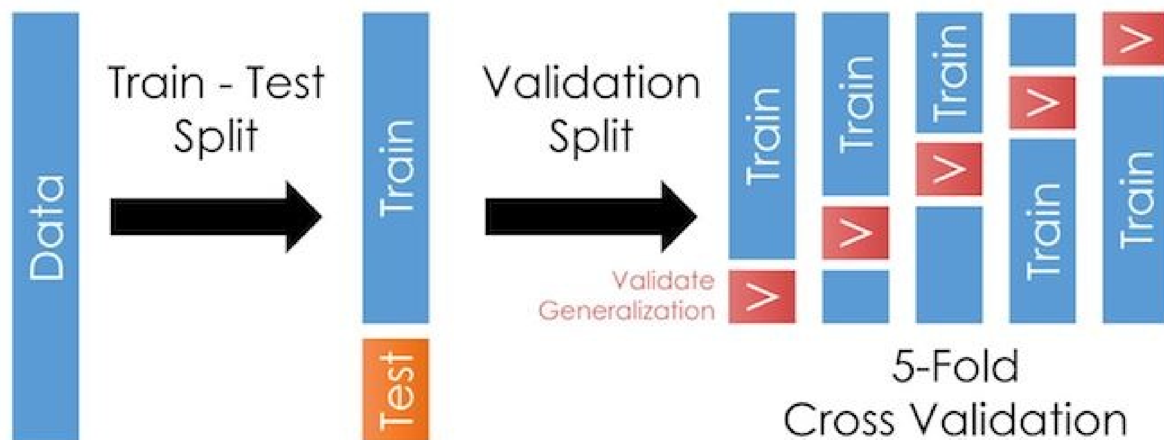
```
Degree 2 polynomial
  Training error: 0.042
Validation error: 0.043
      Test error: 0.063
```

K-Fold Cross-Validation¶

k-fold cross-validation is a common modification of the train-test-split procedure above to reduce variation in validation error. Instead of splitting the original dataset into separate training, validation, and test sets, we create a train-test split.

To compute a model's validation error, we split the training set into k equally-sized subsets: *k folds*. We use one of these folds as the validation set and train the model using the remaining $k-1$ folds. Since there are k folds total, we train the model k times and compute k validation errors. We report the model's final validation error as the average of the k validation errors.

The diagram below illustrates the technique when using five folds.



k-fold cross-validation takes more computation time since we typically have to refit each model from scratch for each fold. However, it computes a more accurate validation error by averaging multiple errors together for each model.

The `scikit-learn` library provides a convenient `sklearn.model_selection.KFold` class to implement k-fold cross-validation.

Summary¶

We use the widely useful cross-validation technique to perform feature and model selection. After computing a train-validation-test split on the original dataset, we use the following procedure to train and choose a model.

1. For each potential set of features, fit a model using the training set. The error of a model on the training set is its *training error*.
2. Check the error of each model on the validation set: its *validation error*. Alternatively, use k-fold cross-validation to compute a validation error. Select the model that achieves the lowest validation error. This is the final choice of features and model.
3. Calculate the *test error*, error of the final model on the test set. This is the final reported

accuracy of the model. We are forbidden from adjusting the features or model to increase test error; doing so effectively converts the test set into a validation set. Instead, we must collect a new test set after making further changes to the features or the model.

The Bias-Variance Tradeoff

Sometimes, we choose a model that is too simple to represent the underlying data generation process. Other times, we choose a model that is too complex—it fits the noise in the data rather than the data's overall pattern.

To understand why this happens, we analyze our models using the tools of probability and statistics. These tools allow us to generalize beyond a few isolated examples to describe fundamental phenomena in modeling. We introduce expectation and variance, then use them to uncover the bias-variance tradeoff.

Show Widgets Open on DataHub

Table of Contents

Expectation and Variance¶

Almost all real-world phenomena contain some degree of randomness, making data generation and collection inherently random processes. Since we fit our models on these data, our models also contain randomness. To represent these random processes mathematically, we use random variables.

Random Variables¶

A **random variable** is an algebraic variable that represents a numerical value determined by a probabilistic event. We typically use capital letters like $ X $ or $ Y $ to denote a random variable.

We must always specify what a given random variable represents. For example, we may write that the random variable $ X $ represents the number of heads in 10 coin flips. The definition of a random variable determines the values it can take on. In this example, $ X $ may only take on values between $ 0 $ and $ 10 $.

We must also be able to determine the probability that the the random variable takes on each value. For example, the probability that $ X = 0 $ is written as $ P(X = 0) = (0.5)^{10} $.

Probability Mass Functions¶

The **probability mass function (PMF)** or the **distribution** of a random variable X yields the probability that X takes on each of its possible values. If we let $\mathbb{X}$ be the set of values that X can take on and x be a particular value in $\mathbb{X}$, the PMF of X must satisfy the following rules:

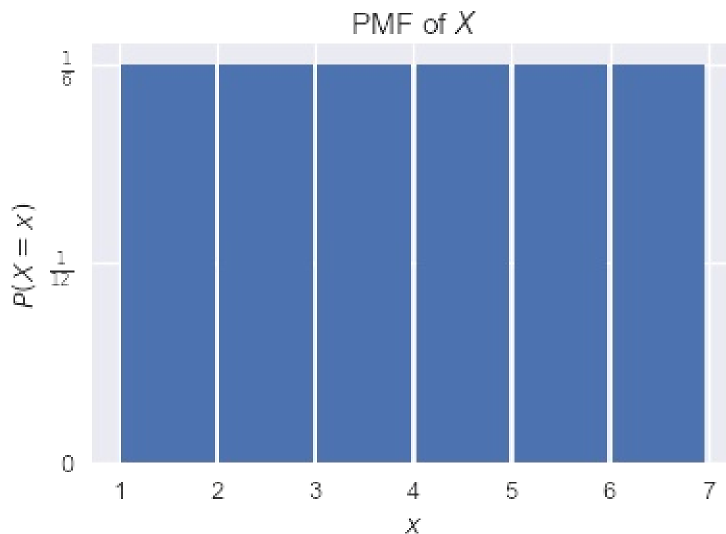
$$ 1) \sum_{x \in \mathbb{X}} \mathbb{P}(X = x) = 1 $$

$$ 2) \text{ For all } x \in \mathbb{X}, 0 \leq \mathbb{P}(X = x) \leq 1 $$

The first rule states that the probabilities for all possible values of $ X $ sum to $ 1 $.

The second rule states that each individual probability for a given value of $ X $ must be between $ 0 $ and $ 1 $.

Suppose we let $ X $ represent the result of one roll from a fair six-sided die. We know that $ X \in \{1, 2, 3, 4, 5, 6\} $ and that $ P(X = 1) = P(X = 2) = \ldots = P(X = 6) = \frac{1}{6} $. We can plot the PMF of $ X $ as a probability distribution:



Expectation¶

We are often interested in the long-run average of a random variable because it gives us a sense of the center of the variable's distribution. We call this long-run average the **expected value**, or **expectation** of a random variable. The expected value of a random variable $ X $ is defined as:

$$\mathbb{E}[X] = \sum_{x\in \mathbb{X}} x \cdot \mathbb{P}(X = x)$$

For example, if $ X $ represents the roll of a single fair six-sided die,

$$ \begin{aligned} \mathbb{E}[X] &= 1 \cdot \mathbb{P}(X = 1) + 2 \cdot \mathbb{P}(X = 2) + \ldots + 6 \cdot \mathbb{P}(X = 6) \\ &= 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} \\ &= 3.5 \end{aligned} $$

Notice that the expected value of $ X $ does not have to be a possible value of $ X $; although in this case $ \mathbb{E}[X] = 3.5 $, $ X $ never takes on the value $ 3.5 $.

Example: Suppose we have a small dataset of four people:

	Age	Name
0	50	Carol
1	52	Bob
2	51	John
3	50	Dave

We pick one person from this dataset uniformly at random. Let $ Y $ be a random variable representing the age of this person. Then:

$$ \begin{aligned} \mathbb{E}[Y] &= 50 \cdot \mathbb{P}(Y = 50) + 51 \cdot \mathbb{P}(Y = 51) + 52 \cdot \mathbb{P}(Y = 52) \\ &= 50 \cdot \frac{2}{4} + 51 \cdot \frac{1}{4} + 52 \cdot \frac{1}{4} \\ &= 50.75 \end{aligned} $$

Example: Suppose we sample two people from this dataset with replacement. If the random variable $ Z $ represents the difference between the ages of the first and second persons in the sample, what is $ \mathbb{E}[Z] $?

To approach this problem, we define two new random variables. We define $ X $ as the age of the first person and $ Y $ as the age of the second. Then, $ Z = X - Y $. Then, we find the joint probability distribution of $ X $ and $ Y $: the probability of each value that $ X $ and $ Y $ can take on simultaneously. For example, the probability that $ X = 51 $ and $ Y = 50 $ is $ P(X = 51, Y = 50) = \frac{1}{4} \cdot \frac{2}{4} = \frac{2}{16} $. In a similar way, we get:

	$Y=50$	$Y=51$	$Y=52$
$X=50$	4/16	2/16	2/16
$X=51$	2/16	1/16	1/16
$X=52$	2/16	1/16	1/16

The above table lets us also find the PMF for $ Z $. For example, $ P(Z = 1) = P(X = 51, Y = 50) + P(X = 52, Y = 51) = \frac{3}{16} $. Thus,

$$ \begin{aligned} \mathbb{E}[Z] &= (-2) \cdot P(Z = -2) + (-1) \cdot P(Z = -1) + \ldots + (2) \cdot P(Z = 2) \\ &= (-2) \cdot \frac{2}{16} + (-1) \cdot \frac{3}{16}+ \ldots + (2) \cdot \frac{2}{16} \\ &= 0 \end{aligned} $$

Since $ \mathbb{E}[Z] = 0 $, we expect that in the long run the difference between the ages of the people in a sample of size 2 will be 0.

Linearity of Expectation¶

When working with linear combinations of random variables as we did above, we can often make good use of the **linearity of expectation** instead of tediously calculating each joint probability individually.

The linearity of expectation states that:

$$ \begin{aligned} \mathbb{E}[X + Y] &= \mathbb{E}[X] + \mathbb{E}[Y] \\ \end{aligned} $$

From this statement we may also derive:

$$ \begin{aligned} \mathbb{E}[cX] &= c\mathbb{E}[X] \\ \end{aligned} $$

where X and Y are random variables, and c is a constant.

In words, the expectation of a sum of any two random variables is equal to the sum of the expectations of the variables. The linearity of expectation holds even if $ X $ and $ Y $ are dependent on each other!

In the previous example, we saw that $ Z = X - Y $. Thus, $ \mathbb{E}[Z] = \mathbb{E}[X - Y] = \mathbb{E}[X] - \mathbb{E}[Y] $.

Now we can calculate $ \mathbb{E}[X] $ and $ \mathbb{E}[Y] $ separately from each other. Since $ \mathbb{E}[X] = \mathbb{E}[Y] = 50.75 $, $ \mathbb{E}[Z] = 50.75 - 50.75 = 0 $.

Note that the linearity of expectation only holds for linear combinations of random variables. For example, $ \mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y] $ is not a linear combination of $ X $ and $ Y $. In this case, $ \mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y] $ is true in general only for independent random variables.

Variance¶

The variance of a random variable is a numerical description of the spread of a random variable. For a random variable $ X $:

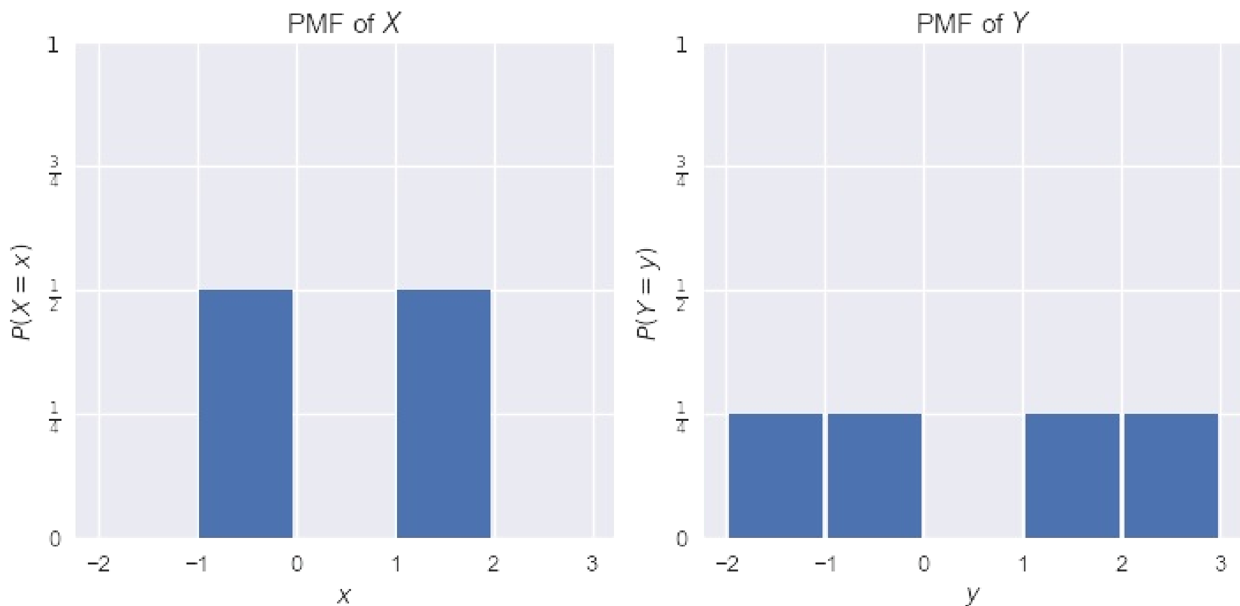
$$ \begin{aligned} Var(X) &= \mathbb{E}[(X - \mathbb{E}[X])^2] \\ \end{aligned} $$

The above formula states that the variance of $ X $ is the average squared distance from $ X $'s expected value.

With some algebraic manipulation that we omit for brevity, we may also equivalently write:

$$ \begin{aligned} Var(X) &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 \\ \end{aligned} $$

Consider the following two random variables $ X $ and $ Y $ with the following probability distributions:



$ X $ takes on values -1 and 1 with probability $ \frac{1}{2} $ each. $ Y $ takes on values -2, -1, 1, and 2 with probability $ \frac{1}{4} $ each. We find that $ \mathbb{E}[X] = \mathbb{E}[Y] = 0 $. Since $ Y $'s distribution has a higher spread than $ X $'s, we expect that $ Var(Y) $ is larger than $ Var(X) $.

$$ \begin{aligned} Var(X) &= \mathbb{E}[(X - \mathbb{E}[X])^2] \\ &= \mathbb{E}[(X - 0)^2] \\ &= \mathbb{E}[X^2] \\ &= (-1)^2 P(X = -1) + (1)^2 P(X = 1) \\ &= 1 \cdot 0.5 + 1 \cdot 0.5 \\ &= 1 \\\\ Var(Y) &= \mathbb{E}[(Y - \mathbb{E}[Y])^2] \\ &= \mathbb{E}[(Y - 0)^2] \\ &= \mathbb{E}[Y^2] \\ &= (-2)^2 P(Y = -2) + (-1)^2 P(Y = -1) + (1)^2 P(Y = 1) + (2)^2 P(Y = 2) \\ &= 4 \cdot 0.25 + 1 \cdot 0.25 + 1 \cdot 0.25 + 4 \cdot 0.25\\ &= 2.5 \end{aligned} $$

As expected, the variance of $ Y $ is greater than the variance of $ X $.

The variance has a useful property to simplify some calculations. If $ X $ is a random variable:

$$ \begin{aligned} Var(aX + b) &= a^2 Var(X) \end{aligned} $$

If two random variables $ X $ and $ Y $ are independent:

$$ \begin{aligned} Var(X + Y) = Var(X) + Var(Y) \end{aligned} $$

Note that the linearity of expectation holds for any $ X $ and $ Y $ even if they are dependent; $ Var(X + Y) = Var(X) + Var(Y) $ holds only when $ X $ and $ Y $ are independent.

Summary¶

In this section, we learn that random variables are variables with multiple possible outcomes. These outcomes must be defined completely and precisely-each outcome must have a well-defined probability of occurrence. Expectation and variance are simple descriptions of a random variable's center and spread. We use the versatility of random variables to describe data generation and modeling.

Show Widgets Open on DataHub

Table of Contents

Risk and Cost Minimization¶

In order to make predictions using data, we define a model, select a cost function, and fit the model's parameters by minimizing the cost. For example, to conduct least squares linear regression, we select the model:

$$ \begin{aligned} f_\hat{\theta} (x) &= \hat{\theta} \cdot x \end{aligned} $$

And the cost function:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y_i - f_\hat{\theta} (X_i))^2\\ \end{aligned} $$

As before, we use $ \hat{\theta} $ as our vector of model parameters, $ x $ as a vector containing a row of a data matrix $ X $, and $ y $ as our vector of observed values to predict. $ X_i $is the i'th row of $ X $ and $ y_i $ is the i'th entry of y.

Observe that our cost function is the average of the loss function values for each row of our data. If we define the squared loss function:

$$ \begin{aligned} \ell(y_i, f_\hat{\theta} (x)) &= (y_i - f_\hat{\theta} (x))^2 \end{aligned} $$

Then we may rewrite our cost function more simply:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i} \ell(y_i, f_\hat{\theta} (X_i)) \end{aligned} $$

The expression above abstracts over the specific loss function; regardless of the loss function we choose, our cost is the average loss.

By minimizing the cost, we select the model parameters that best fit our observed dataset. Thus far, we have refrained from making statements about the population that generated the dataset. In reality, however, we are quite interested in making good predictions on the entire population, not just our data that we have already seen.

Risk¶

If our observed dataset $ X $ and $ y $ are drawn at random from a given population, our observed data are random variables. If our observed data are random variables, our model parameters are also random variables-each time we collect a new set of data and fit a model, the parameters of the model $ f_\hat{\theta} (x) $ will be slightly different.

Suppose we draw one more input-output pair $z, \gamma $ from our population at random. The loss that our model produces on this value is:

$$ \begin{aligned} \ell(\gamma, f_\hat{\theta} (z)) \end{aligned} $$

Notice that this loss is a random variable; the loss changes for different sets of observed data $ X $ and $ y $ and different points $z, \gamma $ from our population.

The **risk** for a model $ f_\hat{\theta} $ is the expected value of the loss above for all training data $ X $, $ y $ and all points $ z$, $ \gamma $ in the population:

$$ \begin{aligned} R(f_\hat{\theta}(x)) = \mathbb{E}[\ell(\gamma, f_\hat{\theta} (z))] \end{aligned} $$

Notice that the risk is an expectation of a random variable and is thus *not* random itself. The expected value of fair six-sided die rolls is 3.5 even though the rolls themselves are random.

The risk above is sometimes called the **true risk** because it tells how a model does on the entire population. If we could compute the true risk for all models, we can simply pick the model with the least risk and know with certainty that the model will perform better in the long run than all other models on our choice of loss function.

Empirical Risk¶

Reality, however, is not so kind. If we substitute in the definition of expectation into the formula for the true risk, we get:

$$ \begin{aligned} R(f_\hat{\theta}) &= \mathbb{E}[\ell(\gamma, f_\hat{\theta} (z))] \\ &= \sum_\gamma \sum_z \ell(\gamma, f_\hat{\theta} (z)) P(\gamma, z) \\ \end{aligned} $$

To further simplify this expression, we need to know $ P(\gamma, z) $, the global probability distribution of observing any point in the population. Unfortunately, this is not so easy. Suppose we are trying to predict the tip amount based on the size of the table. What is the probability that a table of three people gives a tip of $14.50? If we knew the distribution of points exactly, we wouldn't have to collect data or fit a model�we would already know the most likely tip amount for any given table.

Although we do not know the exact distribution of the population, we can approximate it using the observed dataset X and y. If X and y are drawn at random from our population, the distribution of points in X and y is similar to the population distribution. Thus, we treat X and y as our population. Then, the probability that any input-output pair X_i, y_i appear is $\frac{1}{n}$ since each pair appears once out of n points total.

This allows us to calculate the **empirical risk**, an approximation for the true risk:

$$ \begin{aligned} \hat R(f_\hat{\theta}) &= \mathbb{E}[\ell(y_i, f_\hat{\theta} (X_i))] \\ &= \sum_{i=1}^n \ell(y_i, f_\hat{\theta} (X_i)) \frac{1}{n} \\ &= \frac{1}{n} \sum_{i=1}^n \ell(y_i, f_\hat{\theta} (X_i)) \end{aligned} $$

If our dataset is large and the data are drawn at random from the population, the empirical risk $\hat R(f_\hat{\theta})$ is close to the true risk $R(f_\hat{\theta})$. This allows us to pick the model that minimizes the empirical risk.

Notice that this expression is the cost function at the start of the section! By minimizing the average loss, we also minimize the empirical risk. This explains why we often use the average loss as our cost function instead of the maximum loss, for example.

Summary¶

The true risk of a prediction model describes the overall long-run loss that the model will produce for the population. Since we typically cannot calculate the true risk directly, we calculate the empirical risk instead and use the empirical risk to find an appropriate model for prediction. Because the empirical risk is the average loss on the observed dataset, we often minimize the average loss when fitting models.

Show Widgets Open on DataHub

Table of Contents

Model Bias and Variance¶

We have previously seen that our choice of model has two basic sources of error.

Our model may be too simple-a linear model is not able to properly fit data generated from a quadratic process, for example. This type of error arises from model **bias**.

Our model may also fit the random noise present in the data-even if we fit a quadratic process using a quadratic model, the model may predict different outcomes than the true process produces. This type of error arises from model **variance**.

The Bias-Variance Decomposition¶

We can make the statements above more precise by decomposing our formula for model risk. Recall that the **risk** for a model $ f_\hat{\theta} $ is the expected loss for all possible sets of training data $ X $, $ y $ and all input-output points $ z$, $ \gamma $ in the population:

$$ \begin{aligned} R(f_\hat{\theta}) = \mathbb{E}[\ell(\gamma, f_\hat{\theta} (z))] \end{aligned} $$

We denote the process that generates the true population data as $ f_\theta $. Our observed outcomes $ y $ were generated by our population process plus some random noise in data collection: $ y_i = f_\theta(X_i) + \epsilon $, where the random noise $ \epsilon $ is a random variable with a mean of zero: $ \mathbb{E}[\epsilon] = 0 $.

If we use the squared error as our loss function, the above expression becomes:

$$ \begin{aligned} R(f_\hat{\theta}) = \mathbb{E}[(\gamma - f_\hat{\theta} (z))^2] \end{aligned} $$

With some algebraic manipulation that we omit for brevity, we can show that the above expression is equivalent to:

$$ \begin{aligned} R(f_\hat{\theta}) = (\mathbb{E}[f_\hat{\theta}(z)] - f_\theta(z))^2 + \text{Var}(f_\hat{\theta}(z)) + \text{Var}(\epsilon) \end{aligned} $$

The first term in this expression, $ (\mathbb{E}[f_\hat{\theta}(z)] - f_\theta(z))^2 $, is a mathematical expression for the bias of the model. (Technically, this term represents the biased squared, bias^2). The bias is equal to zero if in the long run our choice of model $ f_\hat{\theta}(z) $ predicts the same outcomes produced by the population process $ f_\theta(z) $. The bias is high if our choice of model makes poor predictions of the population process even when we have the entire population as our dataset.

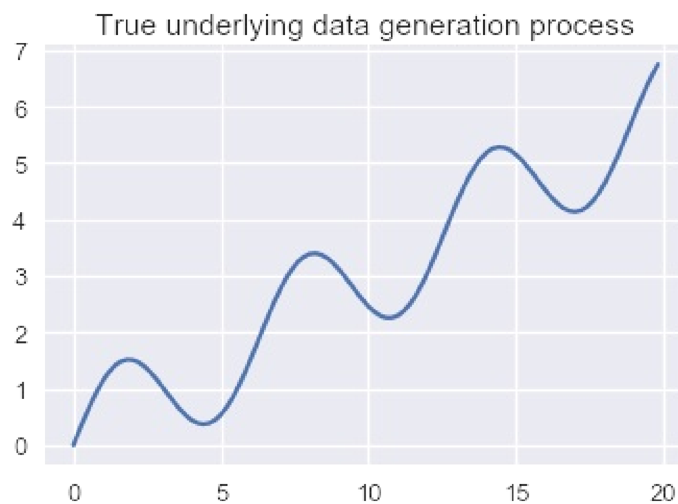
The second term in this expression, $ \text{Var}(f_\hat{\theta}(z)) $, represents the model variance. The variance is low when the model's predictions don't change much when the model is trained on different datasets from the population. The variance is high when the model's predictions change greatly when the model is trained on different datasets from the population.

The third and final term in this expression, $ \text{Var}(\epsilon) $, represents the irreducible error or the noise in the data generation and collection process. This term is small when the data generation and collection process is precise or has low variation. This term is large when the data contain large amounts of noise.

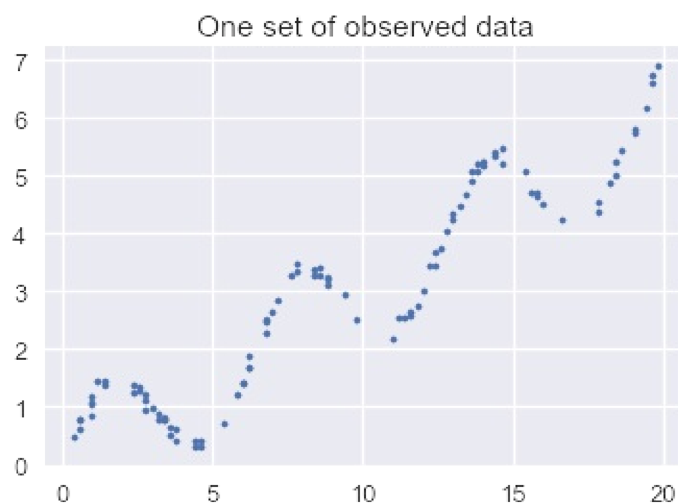
To pick a model that performs well, we seek to minimize the risk. To minimize the risk, we attempt to minimize the bias, variance, and noise terms of the bias-variance decomposition. Decreasing the noise term typically requires improvements to the data collection process�purchasing more precise sensors, for example. To decrease bias and variance, however, we must tune the complexity of our models. Models that are too simple have high bias; models that are too complex have high variance. This is the essence of the *bias-variance tradeoff*, a fundamental issue that we face in choosing models for prediction.

Example: Linear Regression and Sine Waves¶

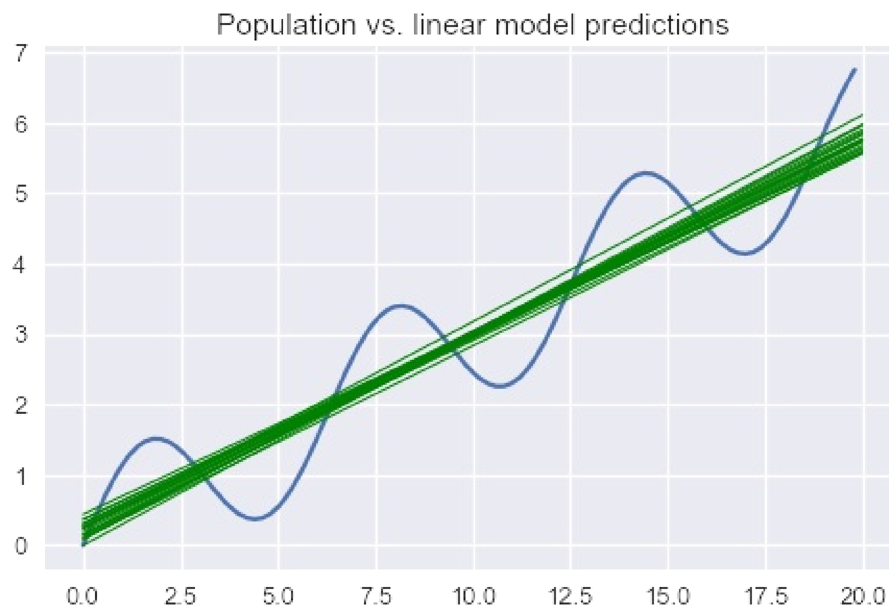
Suppose we are modeling data generated from the oscillating function shown below.



If we randomly draw a dataset from the population, we may end up with the following:



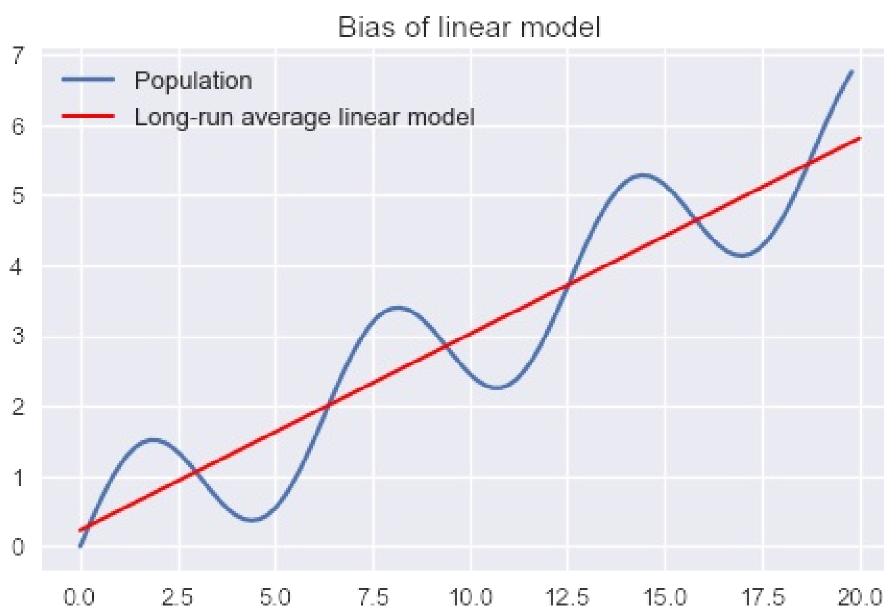
Suppose we draw many sets of data from the population and fit a simple linear model to each one. Below, we plot the population data generation scheme in blue and the model predictions in green.



The plot above clearly shows that a linear model will make prediction errors for this population. We may decompose the prediction errors into bias, variance, and irreducible noise. We illustrate bias of our model by showing that the long-run average linear model will predict different outcomes than the population process:

```
plt.figure(figsize=(8, 5))
xs = np.arange(0, 20, 0.2)
plt.plot(population_x, population_y, label='Population')

plt.plot([avg_line.x_start, avg_line.x_end],
         [avg_line.y_start, avg_line.y_end],
         linewidth=2, c='r',
         label='Long-run average linear model')
plt.title('Bias of linear model')
plt.legend();
```

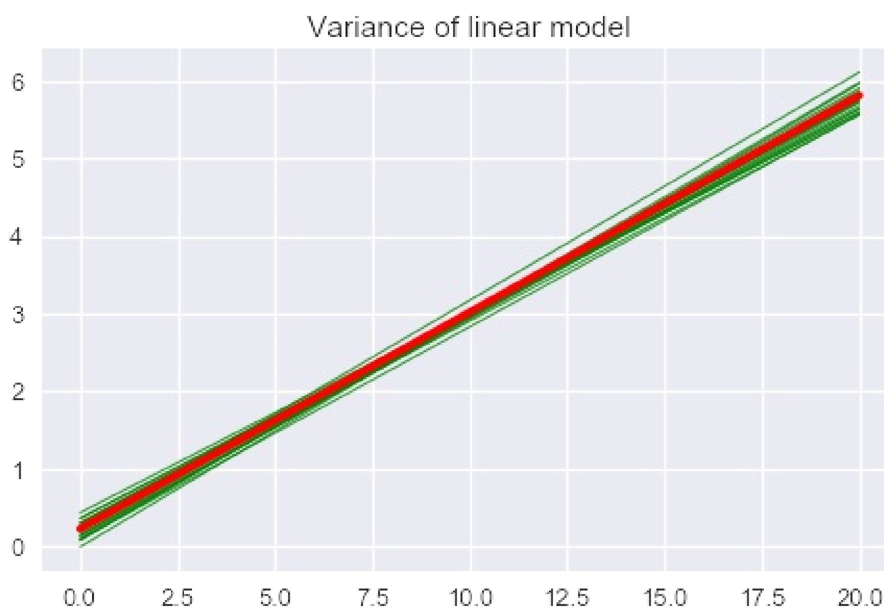


The variance of our model is the variation of the model predictions around the long-run average model:

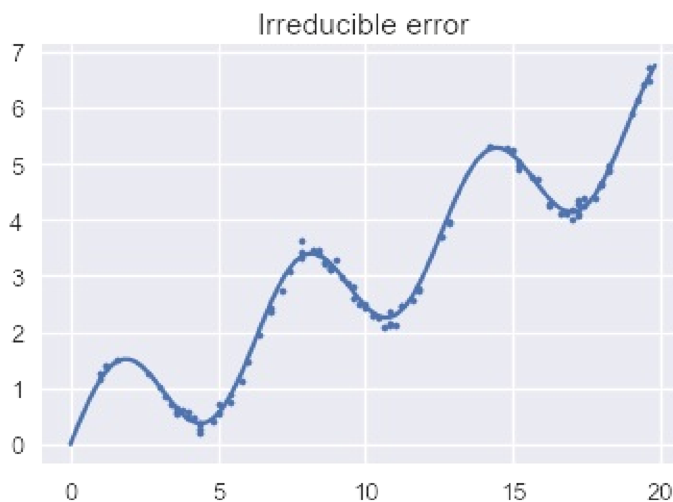
```
plt.figure(figsize=(8, 5))
for x_start, x_end, y_start, y_end in random_lines:
    plt.plot([x_start, x_end], [y_start, y_end], linewidth=1,
c='g', alpha=0.8)

plt.plot([avg_line.x_start, avg_line.x_end],
         [avg_line.y_start, avg_line.y_end],
         linewidth=4, c='r')

plt.title('Variance of linear model');
```



Finally, we illustrate the irreducible error by showing the deviations of the observed points from the underlying population process.



Bias-Variance In Practice¶

In practice, we do not know the population data generation process and thus are unable to precisely determine a model's bias, variance, or irreducible error. Instead, we use our observed dataset as an approximation to the population. As we have seen, however, achieving a low training error does not necessarily mean that our model will have a low test error as well. Fundamentally, this occurs because training error reflects the bias of our model but not the variance; the test error reflects both.

Cross-validation provides a more accurate method of estimating our model error using a single observed dataset by separating data used for training from the data used for model selection and final accuracy. Intuitively, the validation or test error checks the model's performance on a previously unseen dataset and thus allows us to estimate both bias and variance.

Takeaways¶

The bias-variance tradeoff allows us to more precisely describe the modeling phenomena that we have seen thus far.

Underfitting is typically caused by too much bias; overfitting is typically caused by too much variance.

Collecting more data reduces variance. One recent trend is to select a model with low bias and high intrinsic variance (e.g. a neural network) and collect many data points so that the model variance is low enough to make accurate predictions. While effective in practice, collecting enough data for these models tends to require large amounts of time and money.

Collecting more data reduces bias if the model can fit the population process exactly. If the model is inherently incapable of modeling the population (as in the example above), even infinite data cannot get rid of model bias.

Adding a useful feature to the data, such as a quadratic feature when the underlying process is quadratic, reduces bias. Adding a useless feature rarely increases bias.

Adding a feature, whether useful or not, typically increases model variance. In order to increase a model's prediction accuracy, a new feature should decrease bias more than it increases variance.

Summary¶

The bias-variance tradeoff reveals a fundamental problem in modeling. In order to minimize model risk, we use a combination of feature engineering, model selection, and cross-validation to balance bias and variance.

Regularization

Feature engineering can incorporate important information about the data generation process into our model. However, adding features to the data also typically increases the variance of our model and can thus result in worse performance overall. Rather than throwing out features entirely, we can turn to a technique called regularization to reduce the variance of our model while still incorporating as much information about the data as possible.

Show Widgets Open on DataHub

Table of Contents

Regularization Intuition¶

We begin our discussion of regularization with an example that illustrates the importance of regularization.

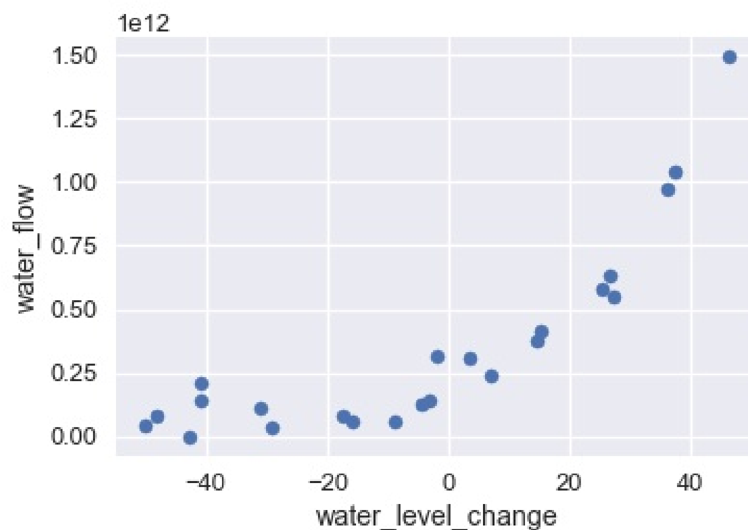
Dam Data¶

The following dataset records the amount of water that flows out of a large dam on a particular day in liters and the amount the water level changed on that day in meters.

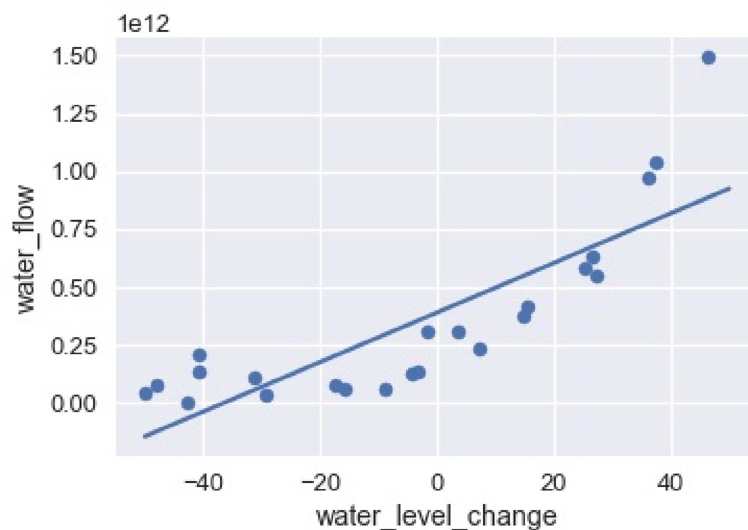
	water_level_change	water_flow
0	-15.936758	6.042233e+10
1	-29.152979	3.321490e+10
2	36.189549	9.727064e+11
...	...	...
20	7.085480	2.363520e+11
21	46.282369	1.494256e+12
22	14.612289	3.781463e+11

23 rows × 2 columns

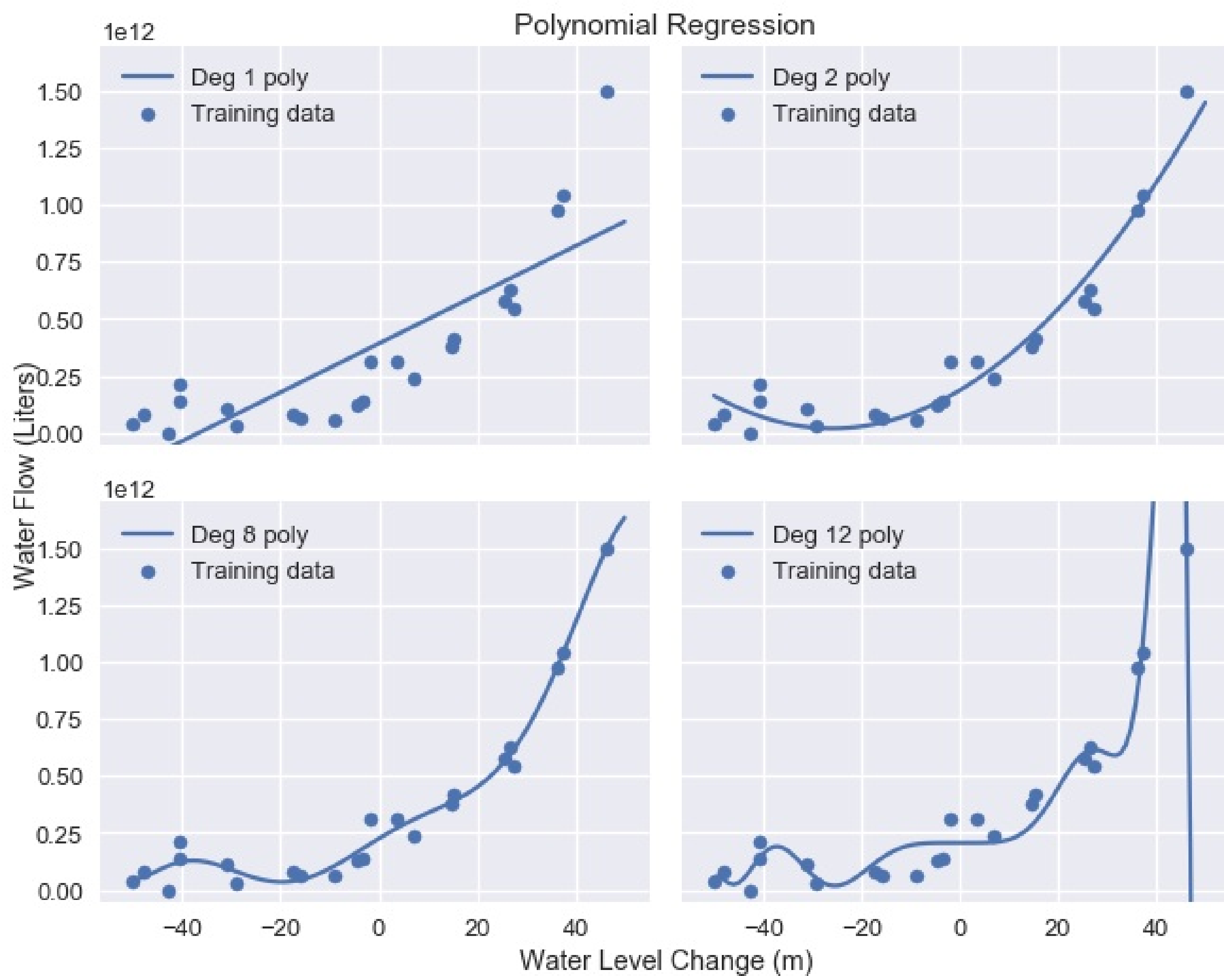
Plotting this data shows an upward trend in water flow as the water level becomes more positive.



To model this pattern, we may use a least squares linear regression model. We show the data and the model's predictions on the plot below.



The visualization shows that this model does not capture the pattern in the data—the model has high bias. As we have previously done, we can attempt to address this issue by adding polynomial features to the data. We add polynomial features of degrees 2, 8, and 12; the chart below shows the training data with each model's predictions.



As expected, the degree 12 polynomial matches the training data well but also seems to fit spurious patterns in the data caused by noise. This provides yet another illustration of the bias-variance tradeoff: the linear model has high bias and low variance while the degree 12 polynomial has low bias but high variance.

Examining Coefficients¶

Examining the coefficients of the degree 12 polynomial model reveals that this model makes predictions according to the following formula:

$$ 207097470825 + 1.8x + 482.6x^2 + 601.5x^3 + 872.8x^4 + 150486.6x^5 \

```
+ 2156.7x^6 - 307.2x^7 - 4.6x^8 + 0.2x^9 + 0.003x^{10} - 0.00005x^{11} + 0x^{12}
```

$$

where $ x $ is the water level change on that day.

The coefficients for the model are quite large, especially for the higher degree terms which contribute significantly to the model's variance ($ x^5 $ and $ x^6 $, for example).

Penalizing Parameters¶

Recall that our linear model makes predictions according to the following, where $ \theta $ is the model weights and $ x $ is the vector of features:

$$ f_\hat{\theta}(x) = \hat{\theta} \cdot x $$

To fit our model, we minimize the mean squared error cost function, where $ X $ is used to represent the data matrix and $ y $ the observed outcomes:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y_i - f_\hat{\theta} (X_i))^2\\ \end{aligned} $$

To minimize the cost above, we adjust $ \hat{\theta} $ until we find the best combination of weights regardless of how large the weights are themselves. However, we have found that larger weights for more complex features result in high model variance. If we could instead alter the cost function to penalize large weight values, the resulting model will have lower variance. We use regularization to add this penalty.

Show Widgets Open on DataHub

Table of Contents

L2 Regularization: Ridge Regression¶

In this section we introduce $ L_2 $ regularization, a method of penalizing large weights in our cost function to lower model variance. We briefly review linear regression, then introduce regularization as a modification to the cost function.

To perform least squares linear regression, we use the model:

$$ f_\hat{\theta}(x) = \hat{\theta} \cdot x $$

We fit the model by minimizing the mean squared error cost function:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}^n(y_i - f_\hat{\theta} (X_i))^2\\ \end{aligned} $$

In the above definitions, $ X $ represents the $ n \times p $ data matrix, $ x $ represents a row of $ X $, $ y $ represents the observed outcomes, and $ \hat{\theta} $ represents the model weights.

L2 Regularization Definition¶

To add $ L_2 $ regularization to the model, we modify the cost function above:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y*i* - *f*\hat{\theta} (X_i))^2

```
+ \lambda \sum_{j = 1}^{p} \hat{\theta_j}^2
```

\end{aligned} $$

Notice that the cost function above is the same as before with the addition of the L_2 regularization $\lambda \sum_{j = 1}^{p} \hat{\theta_j}^2$ term. The summation in this term sums the square of each model weight $\hat{\theta_1}, \hat{\theta_2}, \ldots, \hat{\theta_p}$. The term also introduces a new scalar model parameter λ that adjusts the regularization penalty.

The regularization term causes the cost to increase if the values in $\hat{\theta}$ are further away from 0. With the addition of regularization, the optimal model weights minimize the combination of loss and regularization penalty rather than the loss alone. Since the resulting model weights tend to be smaller in absolute value, the model has lower variance and higher bias.

Using L_2 regularization with a linear model and the mean squared error cost function is also known more commonly as **ridge regression**.

The Regularization Parameter

The regularization parameter λ controls the regularization penalty. A small λ results in a small penalty—if $\lambda = 0$ the regularization term is also 0 and the cost is not regularized at all.

A large λ terms results in a large penalty and therefore a simpler model. Increasing λ decreases the variance and increases the bias of the model. We use cross-validation to select the value of λ that minimizes the validation error.

Note about regularization in `scikit-learn`:

`scikit-learn` provides regression models that have regularization built-in. For example, to conduct ridge regression you may use the `sklearn.linear_model.Ridge` regression model. Note that `scikit-learn` models call the regularization parameter `alpha` instead of λ.

`scikit-learn` conveniently provides regularized models that perform cross-validation to select a good value of λ. For example, the `sklearn.linear_model.RidgeCV` allows users to input regularization parameter values and will automatically use cross-validation to select the parameter value with the least validation error.

Bias Term Exclusion

Note that the bias term θ_0 is not included in the summation of the regularization term. We do not penalize the bias term because increasing the bias term does not increase the variance of our model—the bias term simply shifts all predictions by a constant value.

Data Normalization¶

Notice that the regularization term $\lambda \sum_{j = 1}^{p} \hat{\theta_j}^2$ penalizes each $\hat{\theta_j}$ equally. However, the effect of each $\hat{\theta_j}$ differs depending on the data itself. Consider this section of the water flow dataset after adding degree 8 polynomial features:

	deg_0_feat	deg_1_feat	...	deg_6_feat	deg_7_feat
0	-15.94	253.98	...	-261095791.08	4161020472.12
1	-29.15	849.90	...	-17897014961.65	521751305227.70
2	36.19	1309.68	...	81298431147.09	2942153527269.12
3	37.49	1405.66	...	104132296999.30	3904147586408.71
4	-48.06	2309.65	...	-592123531634.12	28456763821657.78

5 rows × 8 columns

We can see that the degree 7 polynomial features have much larger values than the degree 1 features. This means that a large model weight for the degree 7 features affects the predictions much more than a large model weight for the degree 1 features. If we apply regularization to this data directly, the regularization penalty will disproportionately lower the model weight for the lower degree features. In practice, this often results in high model variance even after applying regularization since the features with large effect on prediction will not be affected.

To combat this, we *normalize* each data column by subtracting the mean and scaling the values in each column to be between -1 and 1. In `scikit-learn`, most regression models allow initializing with `normalize=True` to normalize the data before fitting.

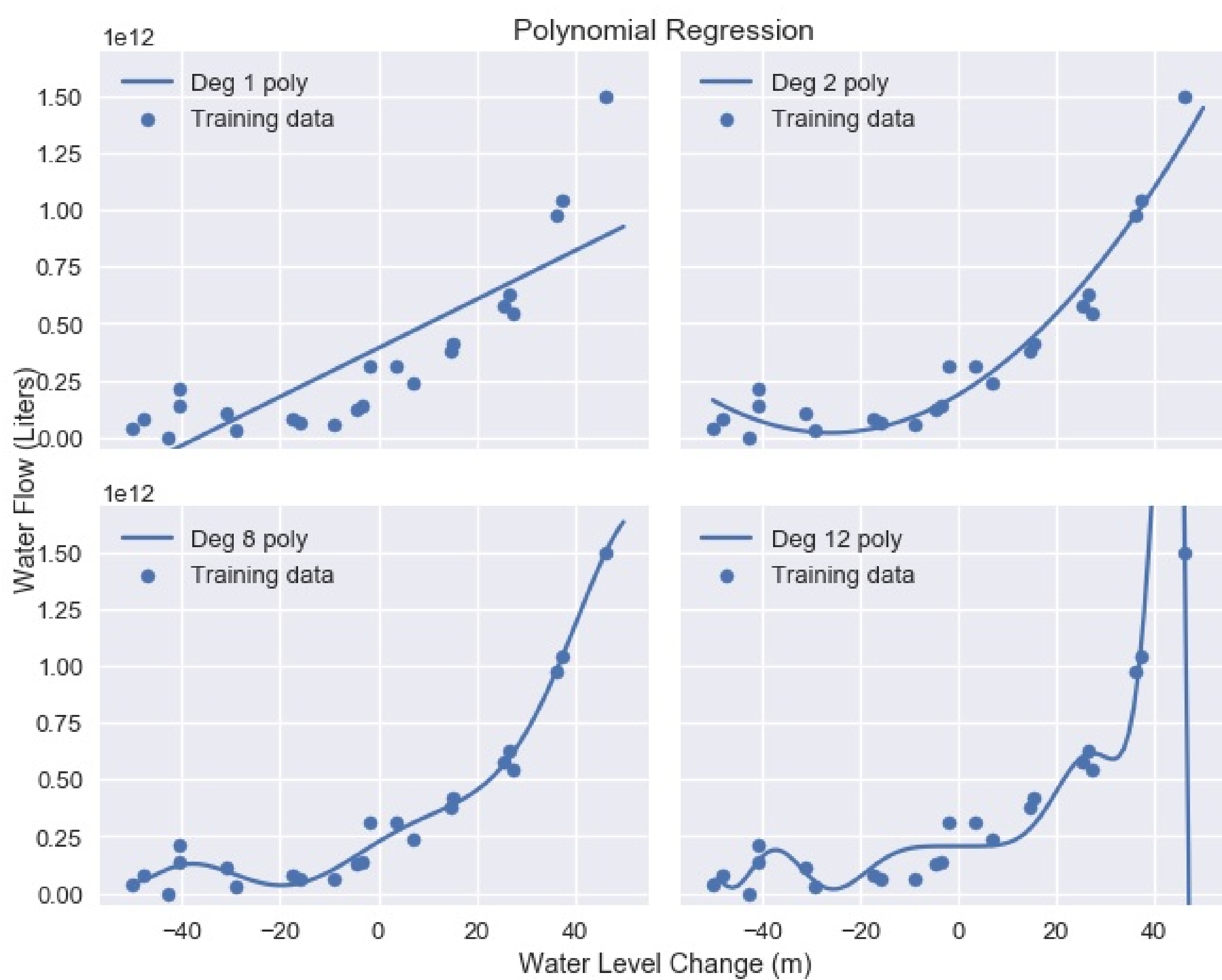
Another analogous technique is *standardizing* the data columns by subtracting the mean and dividing by the standard deviation for each data column.

Using Ridge Regression¶

We have previously used polynomial features to fit polynomials of degree 2, 8, and 12 to water flow data. The original data and resulting model predictions are repeated below.

	water_level_change	water_flow
0	-15.94	60422330445.52
1	-29.15	33214896575.60
2	36.19	972706380901.06
...	...	...
20	7.09	236352046523.78
21	46.28	1494256381086.73
22	14.61	378146284247.97

23 rows × 2 columns



To conduct ridge regression, we first extract the data matrix and the vector of outcomes from the data:

```
X = df.iloc[:, [0]].as_matrix()
y = df.iloc[:, 1].as_matrix()
print('X: ')
print(X)
print()
print('y: ')
print(y)
```

```
X:
[[-15.94]
 [-29.15]
 [ 36.19]
 ...
 [  7.09]
 [ 46.28]
 [ 14.61]]

y:
[6.04e+10 3.32e+10 9.73e+11 ... 2.36e+11 1.49e+12 3.78e+11]
```

Then, we apply a degree 12 polynomial transform to `X`:

```
from sklearn.preprocessing import PolynomialFeatures

# We need to specify include_bias=False since sklearn's
classifiers
# automatically add the intercept term.
X_poly_8 = PolynomialFeatures(degree=8,
include_bias=False).fit_transform(X)
print('First two rows of transformed X:')
print(X_poly_8[0:2])
```

```
First two rows of transformed X:
[[-1.59e+01  2.54e+02 -4.05e+03  6.45e+04 -1.03e+06  1.64e+07
-2.61e+08
   4.16e+09]
 [-2.92e+01  8.50e+02 -2.48e+04  7.22e+05 -2.11e+07  6.14e+08
-1.79e+10
   5.22e+11]]
```

We specify `alpha` values that `scikit-learn` will select from using cross-validation, then use the `RidgeCV` classifier to fit the transformed data.

```
from sklearn.linear_model import RidgeCV

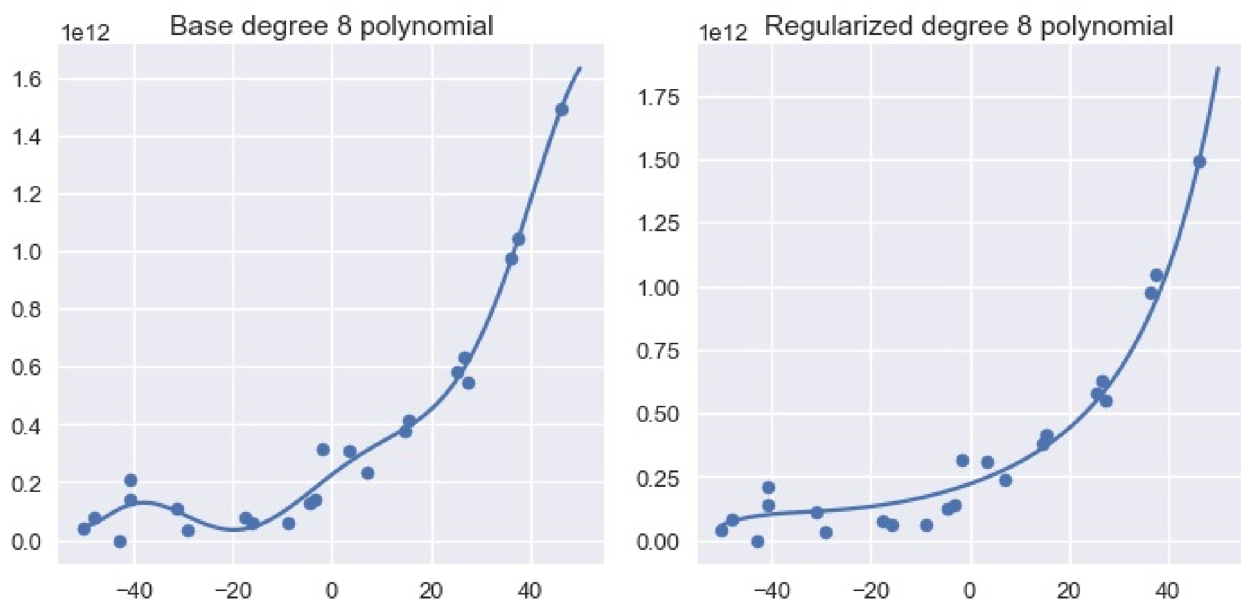
alphas = [0.01, 0.1, 1.0, 10.0]

# Remember to set normalize=True to normalize data
clf = RidgeCV(alphas=alphas, normalize=True).fit(X_poly_8, y)

# Display the chosen alpha value:
clf.alpha_
```

```
0.1
```

Finally, we plot the model predictions for the base degree 8 polynomial classifier next to the regularized degree 8 classifier:

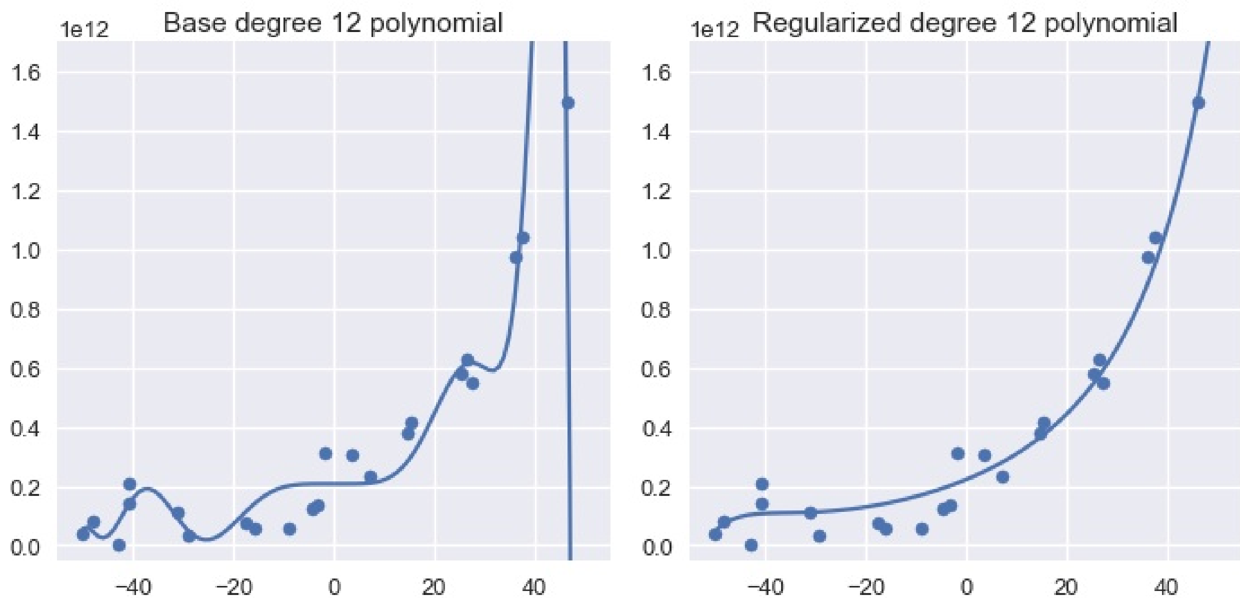


We can see that the regularized polynomial is smoother than the base degree 8 polynomial and still captures the major trend in the data.

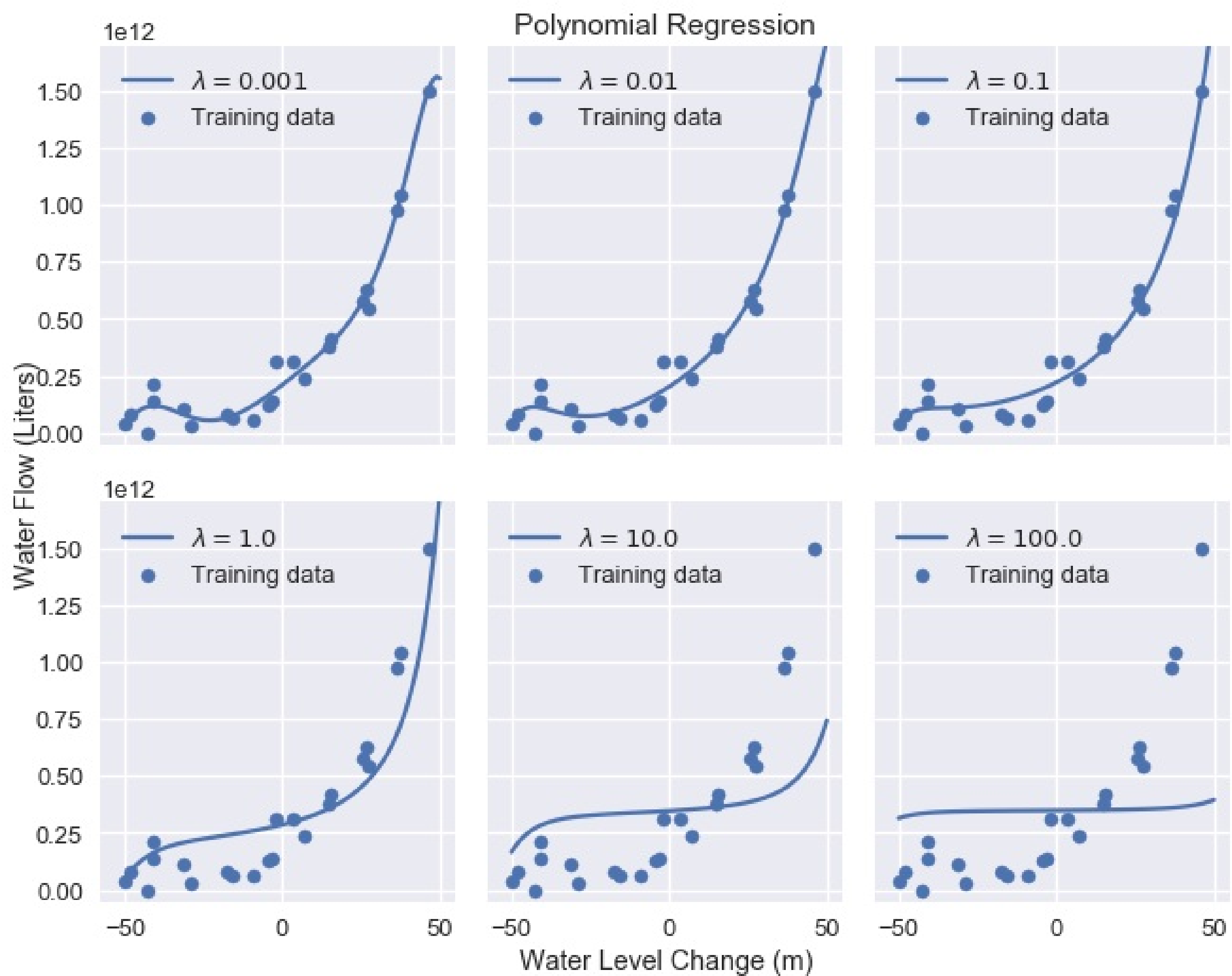
Comparing the coefficients of the non-regularized and regularized models shows that ridge regression favors placing model weights on the lower degree polynomial terms:

	Base	Regularized
degree		
0	225782472111.94	221063525725.23
1	13115217770.78	6846139065.96
2	-144725749.98	146158037.96
3	-10355082.91	1930090.04
4	567935.23	38240.62
5	9805.14	564.21
6	-249.64	7.25
7	-2.09	0.18
8	0.03	0.00

Repeating the process for degree 12 polynomial features results in a similar result:



Increasing the regularization parameter results in progressively simpler models. The plot below demonstrates the effects of increasing the regularization amount from 0.001 to 100.0.



As we can see, increasing the regularization parameter increases the bias of our model. If our parameter is too large, the model becomes a constant model because any non-zero model weight is heavily penalized.

Summary¶

Using $ L_2 $ regularization allows us to tune model bias and variance by penalizing large model weights. $ L_2 $ regularization for least squares linear regression is also known by the more common name ridge regression. Using regularization adds an additional model parameter $ \lambda $ that we adjust using cross-validation.

Show Widgets Open on DataHub

Table of Contents

L1 Regularization: Lasso Regression¶

In this section we introduce $ L_1 $ regularization, another regularization technique that is useful for feature selection.

We start with a brief review of $ L_2 $ regularization for linear regression. We use the model:

$$ f_\hat{\theta}(x) = \hat{\theta} \cdot x $$

We fit the model by minimizing the mean squared error cost function with an additional regularization term:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y*i* - *f*\hat{\theta} (X_i))^2

```
+ \lambda \sum_{j = 1}^{p} \hat{\theta_j}^2
```

\end{aligned} $$

In the above definitions, $ X $ represents the $ n \times p $ data matrix, $ x $ represents a row of $ X $, $ y $ represents the observed outcomes, $ \hat{\theta} $ represents the model weights, and $ \lambda $ represents the regularization parameter.

L1 Regularization Definition¶

To add $ L_1 $ regularization to the model, we modify the cost function above:

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y*i* - *f*\hat{\theta} (X_i))^2

```
+ \lambda \sum_{j = 1}^{p} |\hat{\theta_j}|
```

\end{aligned} $$

Observe that the two cost functions only differ in their regularization term. $ L_1 $ regularization penalizes the sum of the absolute weight values instead of the sum of squared values.

Using $ L_1 $ regularization with a linear model and the mean squared error cost function is also known more commonly as **lasso regression**. (Lasso stands for Least Absolute Shrinkage and Selection Operator.)

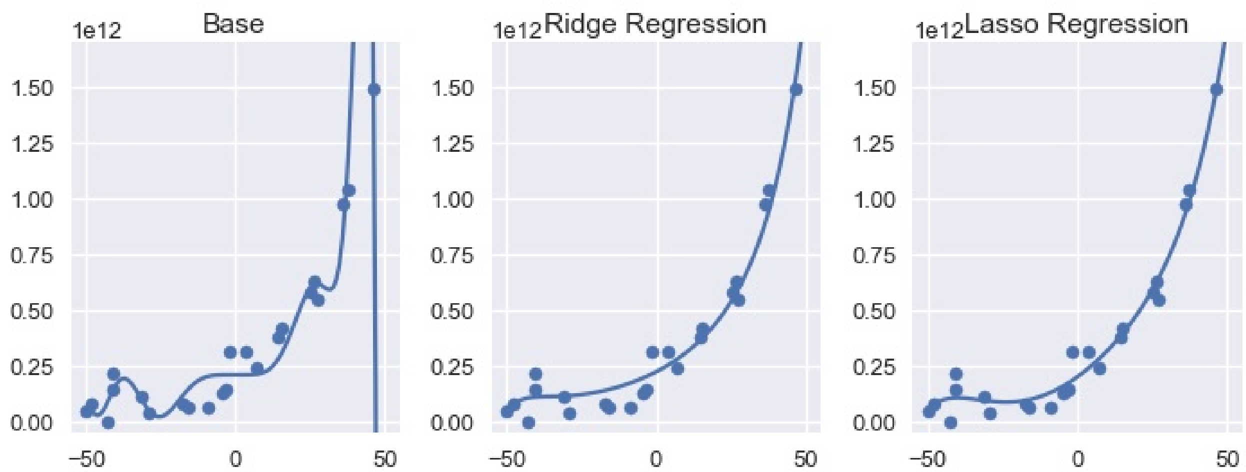
Comparing Lasso and Ridge Regression¶

To conduct lasso regression, we make use of `scikit-learn`'s convenient `LassoCV` classifier, a version of the `Lasso` classifier that performs cross-validation to select the regularization parameter. Below, we display our dataset of water level change and water flow out of a dam.

	water_level_change	water_flow
0	-15.94	60422330445.52
1	-29.15	33214896575.60
2	36.19	972706380901.06
...	...	...
20	7.09	236352046523.78
21	46.28	1494256381086.73
22	14.61	378146284247.97

23 rows � 2 columns

Since the procedure is almost identical to using the `RidgeCV` classifier from the previous section, we omit the code and instead display the base degree 12 polynomial, ridge regression, and lasso regression model predictions below.



We can see that both regularized models have less variance than the base degree 12 polynomial. At a glance, it appears that using $ L_2 $ and $ L_1 $ regularization produces nearly identical models. Comparing the coefficients of ridge and lasso regression, however, reveals the most significant difference between the two types of regularization: the lasso regression model sets a number of model weights to zero.

	Ridge	Lasso
degree		
0	221303288116.2275085449	198212062407.2835693359
1	6953405307.7653837204	9655088668.0876655579
2	142621063.9297277331	198852674.1646585464
3	1893283.0567885502	0.0000000000
4	38202.1520293704	34434.3458919188
5	484.4262914111	975.6965959434
6	8.1525126516	0.0000000000
7	0.1197232472	0.0887942172
8	0.0012506185	0.0000000000
9	0.0000289599	0.0000000000
10	-0.0000000004	0.0000000000
11	0.0000000069	0.0000000000
12	-0.0000000001	-0.0000000000

If you will forgive the verbose output above, you will notice that ridge regression results in non-zero weights for the all polynomial features. Lasso regression, on the other hand, produces weights of zero for seven features.

In other words, the lasso regression model completely tosses out a majority of the features when making predictions. Nonetheless, the plots above show that the lasso regression model will make nearly identical predictions compared to the ridge regression model.

Feature Selection with Lasso Regression¶

Lasso regression performs **feature selection�**it discards a subset of the original features when fitting model parameters. This is particularly useful when working with high-dimensional data with many features. A model that only uses a few features to make a prediction will run much faster than a model that requires many calculations. Since unneeded features tend to increase model variance without decreasing bias, we can sometimes increase the accuracy of other models by using lasso regression to select a subset of features to use.

Lasso vs. Ridge In Practice¶

If our goal is merely to achieve the highest prediction accuracy, we can try both types of regularization and use cross-validation to select between the two types.

Sometimes we prefer one type of regularization over the other because it maps more closely to the domain we are working with. For example, if know that the phenomenon we are trying to model results from many small factors, we might prefer ridge regression because it won't discard these factors. On the other hand, some outcomes result from a few highly influential features. We prefer lasso regression in these situations because it will discard unneeded features.

Lasso regression models tend to take more computation to fit than ridge regression models which may also be a concern depending on your data and your available computational resources.

Summary¶

Using L_1 regularization, like L_2 regularization, allows us to tune model bias and variance by penalizing large model weights. L_1 regularization for least squares linear regression is also known by the more common name lasso regression. Lasso regression may also be used to perform feature selection since it discards insignificant features.

Classification

Thus far we have studied models for regression, the process of making continuous, numerical estimations based on data. We now turn our attention to **classification**, the process of making categorical predictions based on data. For example, weather stations are interested in predicting whether tomorrow will be rainy or not using the weather conditions today.

Together, regression and classification compose the primary approaches for *supervised learning*, the general task of learning a model based on observed input-output pairs.

We may reconstruct classification as a type of regression problem. Instead of creating a model to predict an arbitrary number, we create a model to predict a probability that a data point belongs to a category. This allows us to reuse the machinery of linear regression for a regression on probabilities: logistic regression.

Show Widgets Open on DataHub

Table of Contents

Regression on Probabilities¶

In basketball, players score by shooting a ball through a hoop. One such player, LeBron James, is widely considered one of the best basketball players ever for his incredible ability to score.


CAVS
23

LeBron plays in the National Basketball Association (NBA), the United States's premier basketball league. We've collected a dataset of all of LeBron's attempts in the 2017 NBA Playoff Games using the NBA statistics website (https://stats.nba.com/).

```
lebron = pd.read_csv('lebron.csv')
lebron
```

	game_date	minute	opponent	action_type	shot_type	shot_dista
0	20170415	10	IND	Driving Layup Shot	2PT Field Goal	0
1	20170415	11	IND	Driving Layup Shot	2PT Field Goal	0
2	20170415	14	IND	Layup Shot	2PT Field Goal	0
...	...	...	...	...	...	...
381	20170612	46	GSW	Driving Layup Shot	2PT Field Goal	1
382	20170612	47	GSW	Turnaround Fadeaway shot	2PT Field Goal	14
383	20170612	48	GSW	Driving Layup Shot	2PT Field Goal	2

384 rows × 7 columns

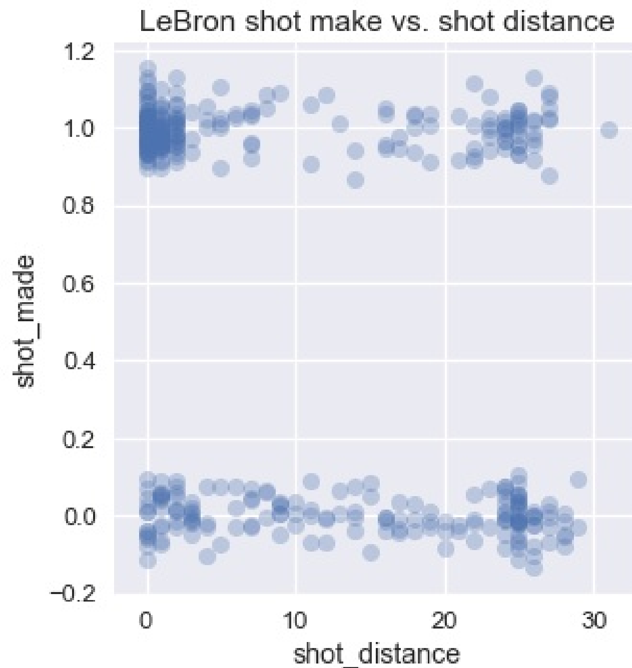
This dataset contains one row containing the following attributes of every shot LeBron attempted:

- `game_date` : The date of the game played.
- `minute` : The minute that the shot was attempted (each NBA game is 48 minutes long).
- `opponent` : The team abbreviation of LeBron's opponent.
- `action_type` : The type of action leading up to the shot.
- `shot_type'` : The type of shot (either a 2 point shot or 3 point shot).
- `shot_distance` : The distance from the basket when shot was attempted.
- `shot_made` : `0` if the shot missed, `1` if the shot went in.

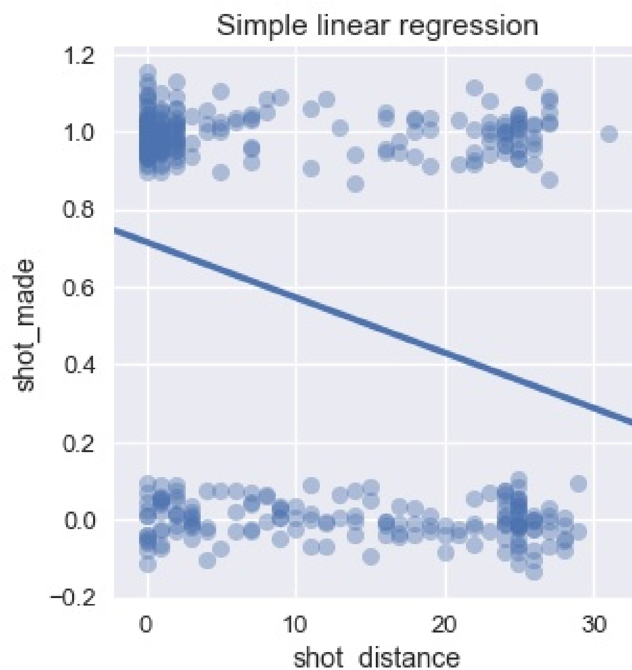
We would like to use this dataset to predict whether LeBron will make future shots. This is a *classification problem*; we predict a category, not a continuous number as we do in regression.

We may reframe this classification problem as a type of regression problem by predicting the *probability* that a shot will go in. For example, we expect that the probability that LeBron makes a shot is lower when he is farther away from the basket.

We plot the shot attempts below, showing the distance from the basket on the x-axis and whether he made the shot on the y-axis. We've jittered the points slightly on the y-axis to mitigate overplotting.

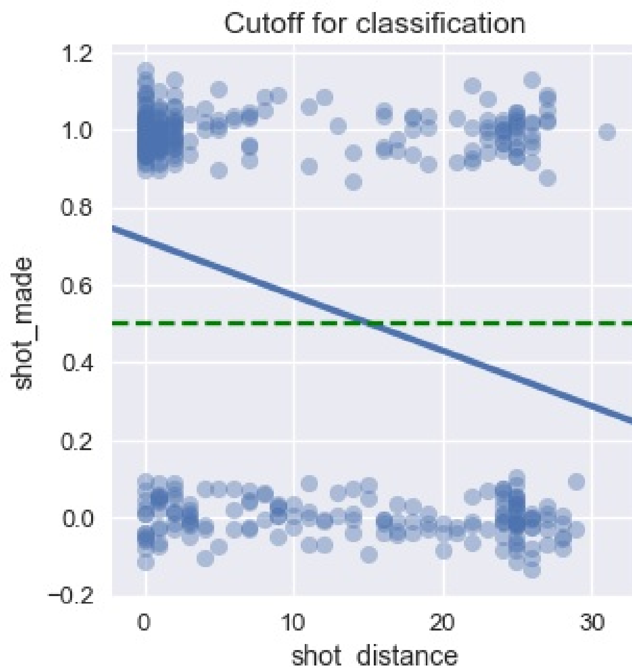


We can see that LeBron tends to make most shots when he is within five feet of the basket. A simple least squares linear regression model fit on this data produces the following predictions:



This regression predicts a continuous value. To perform classification, however, we need to convert this value into a category: a shot make or a miss. We can accomplish this by setting a cutoff. If the regression predicts a value greater than 0.5, we predict that the shot will make. Otherwise, we predict that the shot will miss.

We draw the cutoff below as a green dashed line. According to this cutoff, our model predicts that LeBron will make a shot if he is closer than 15 feet away from the basket.



In the steps above, we attempt to perform a regression to predict the probability that a shot will go in. If our regression produces a probability, setting a cutoff of 0.5 means that we predict that a shot will go in when our model thinks the shot going in is more likely than the shot missing.

Issues with Linear Regression for Probabilities¶

Unfortunately, our linear model's predictions cannot be interpreted as probabilities. Valid probabilities must lie between zero and one; our linear model violates this condition. For example, the probability that LeBron makes a shot when he is 100 feet away from the basket should be close to zero. In this case, however, our model will predict a negative value.

If we alter our regression model so that its predictions may be interpreted as probabilities, we have no qualms about using its predictions for classification. We accomplish this with a new prediction function and a new cost function. The resulting model is called a **logistic regression** model.

Show Widgets Open on DataHub

Table of Contents

The Logistic Model¶

In this section, we introduce the **logistic model**, a regression model that we use to predict probabilities.

Recall that fitting a model for prediction requires three components: a model that makes predictions, a cost function, and an optimization method. For the by-now familiar least squares linear regression, we select the model:

$$ \begin{aligned} f_\hat{\theta} (x) &= \hat{\theta} \cdot x \end{aligned} $$

And the cost function:

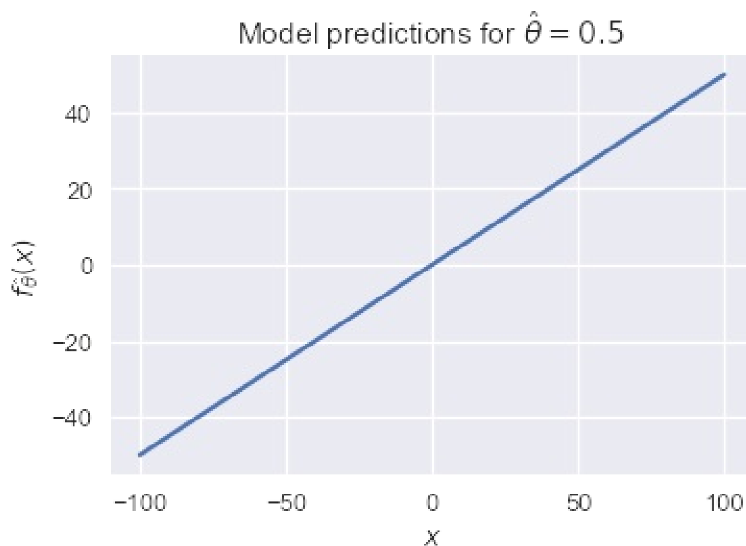
$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_{i}(y_i - f_\hat{\theta} (X_i))^2\\ \end{aligned} $$

We use gradient descent as our optimization method. In the definitions above, $ X $ represents the $ n \times p $ data matrix, $ x $ represents a row of $ X $, $ y $ represents the observed outcomes, and $ \hat{\theta} $ represents the model weights.

Real Numbers to Probabilities¶

Observe that the model $ f_\hat{\theta} (x) = \hat{\theta} \cdot x $ can output any real number $ \mathbb{R} $ since it produces a linear combination of the values in the vector $ x $ which itself can contain any value from $ \mathbb{R} $.

We can easily visualize this when $ x $ is a scalar. If $ \hat \theta = 0.5$, our model becomes $ f_\hat{\theta} (x) = 0.5 x $. Its predictions can take on any value from negative infinity to positive infinity:



For classification tasks, we want to constrain $ f_\hat{\theta}(x) $ so that its output can be interpreted as a probability. This means that it may only output values in the range $ [0, 1] $. In addition, we would like large values of $ f_\hat{\theta}(x) $ to correspond to high probabilities and small values to low probabilities.

The Logistic Function¶

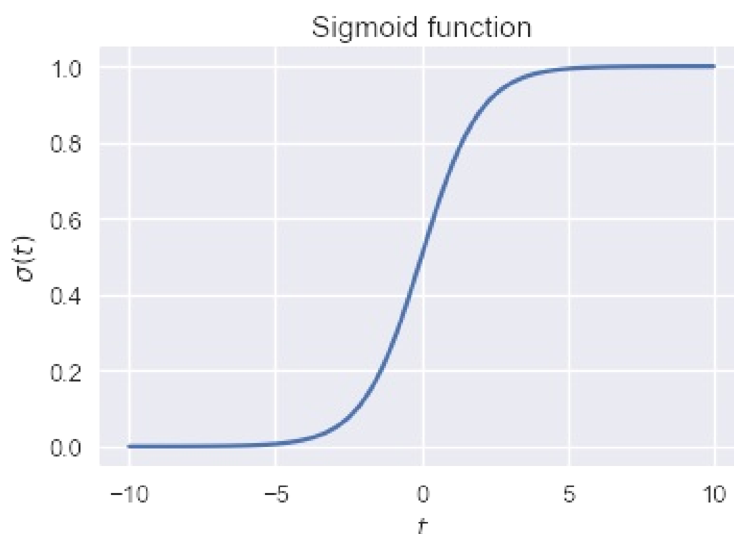
To accomplish this, we introduce the **logistic function**, often called the **sigmoid function**:

$$ \begin{aligned} \sigma(t) = \frac{1}{1 + e^{-t}} \end{aligned} $$

For ease of reading, we often replace $ e^x $ with $ \text{exp}(x) $ and write:

$$ \begin{aligned} \sigma (t) = \frac{1}{1 + \text{exp}(-t)} \end{aligned} $$

We plot the sigmoid function for values of $ t \in [-10, 10] $ below.



Observe that the sigmoid function $ \sigma(t) $ takes in any real number $ \mathbb{R} $ and outputs only numbers between 0 and 1. The function is monotonically increasing on its input $ t $; large values of $ t $ correspond to values closer to 1, as desired. This is not a coincidence—the sigmoid function may be derived from a log ratio of probabilities, although we omit the derivation for brevity.

Logistic Model Definition¶

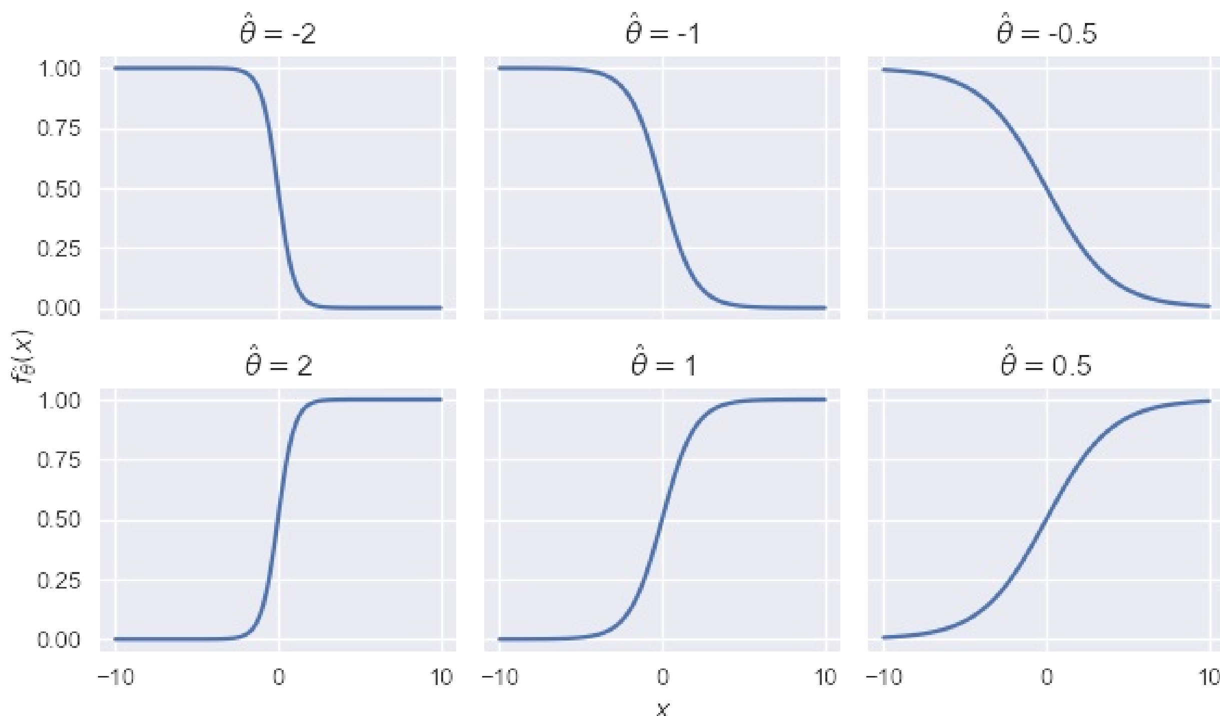
We may now take our linear model $ f_\hat{\theta} (x) = \hat{\theta} \cdot x $ and use it as the input to the sigmoid function to create the model:

$$ \begin{aligned} f_\hat{\theta} (x) = \sigma(\hat{\theta} \cdot x) \end{aligned} $$

In other words, we take the output of linear regression—any number in $ \mathbb{R} $— and use the sigmoid function to restrict the model's final output to be a valid probability between zero and one.

This model is called the **logistic model**.

To develop some intuition for how the model behaves, we restrict $ x $ to be a scalar and plot the logistic model's output for several values of $ \hat{\theta} $.



We see that changing $ \hat{\theta} $ changes the sharpness of the curve; the further away from $ 0 $, the sharper the curve.

Summary¶

We introduce the logistic model, a new prediction function that outputs probabilities. To construct the model, we use the output of linear regression as the input of the logistic function.

Show Widgets Open on DataHub

Table of Contents

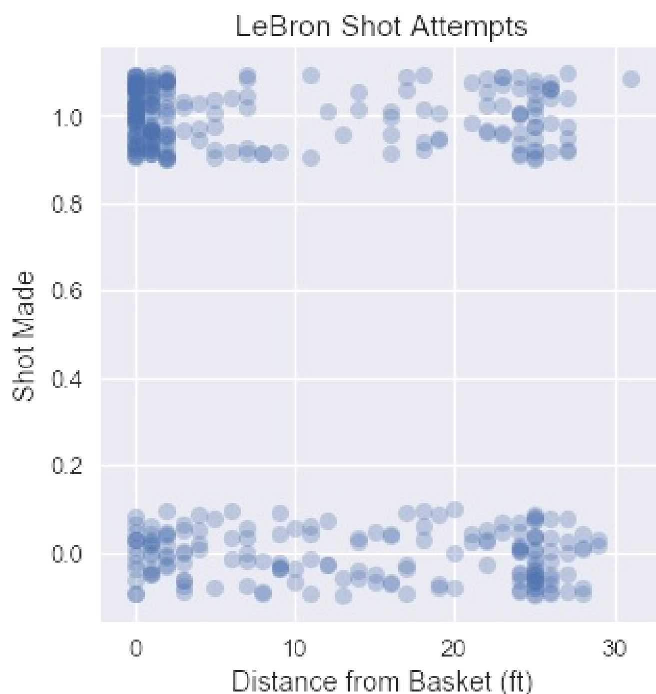
A Cost Function for the Logistic Model¶

We have defined a regression model for probabilities, the logistic model:

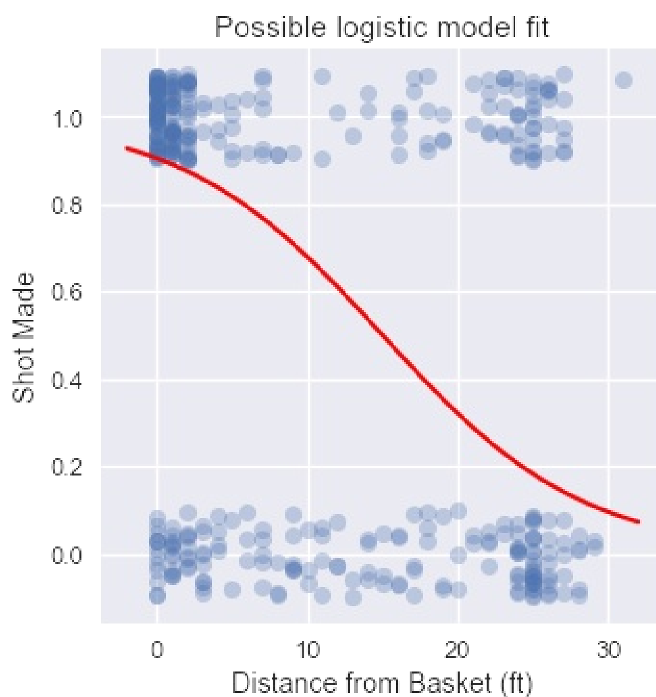
$$ \begin{aligned} f_\hat{\theta} (x) = \sigma(\hat{\theta} \cdot x) \end{aligned} $$

Like the model for linear regression, this model has parameters $ \hat{\theta} $, a vector that contains one parameter for each feature of $ x $. We now address the problem of defining a cost function for this model which allows us to fit the model's parameters to data.

Intuitively, we want the model's predictions to match the data as closely as possible. Below we recreate a plot of LeBron's shot attempts in the 2017 NBA Playoffs using the distance of each shot from the basket. The points are jittered on the y-axis to mitigate overplotting.



Noticing the large cluster of made shots close to the basket and the smaller cluster of missed shots further from the basket, we expect that a logistic model fitted on this data might look like:



Although we can use the mean squared error cost function as we have for linear regression, for a logistic model this cost function is non-convex and thus difficult to optimize.

The Cross Entropy Cost¶

Instead of the mean squared error, we use the **cross entropy cost**. Let $ X $ represent the $ n \times p $ input data matrix, $ y $ the vector of observed data values, and $ f_\hat{\theta}(x) $ the logistic model. Using this notation, the cross entropy cost is defined as:

$$ \begin{aligned} L(\hat{\theta}, X, y) = \frac{1}{n} \sum_i \left(- y_i \ln (f_\hat{\theta}(X_i)) - (1 - y_i) \ln (1 - f_\hat{\theta}(X_i) \right) \end{aligned} $$

You may observe that as usual we take the mean loss over each point in our dataset. This loss is called the cross entropy loss and forms the inner expression in the above summation:

$$ \begin{aligned} \ell(\hat{\theta}, X, y) = - y_i \ln (f_\hat{\theta}(X_i)) - (1 - y_i) \ln (1 - f_\hat{\theta}(X_i)) \end{aligned} $$

Recall that each $ y_i $ is either 0 or 1 in our dataset. If $ y_i = 0 $, the first term in the loss is zero. If $ y_i = 1 $, the second term in the loss is zero. Thus, only one term of the cross entropy loss contributes to the overall cost for each point in our dataset.

Suppose $ y_i = 0 $ and our predicted probability $ f_\hat{\theta}(X_i) = 0 $—our model is completely correct. The loss for this point will be:

$$ \begin{aligned} \ell(\hat{\theta}, X, y) &= - y_i \ln (f_\hat{\theta}(X_i)) - (1 - y_i) \ln (1 - f_\hat{\theta}(X_i)) \\ &= - 0 - (1 - 0) \ln (1 - 0) \\ &= - \ln (1) \\ &= 0 \end{aligned} $$

As expected, the loss for a correct prediction is 0. You may verify that the further the predicted probability is from the true value, the greater the loss.

Minimizing the overall cross entropy cost requires the model $f_\hat{\theta}(x)$ to make the most accurate predictions it can. Conveniently, this cost function is convex, making gradient descent a useful choice for optimization.

Statistical justification for the cross entropy cost

The cross entropy cost also has fundamental underpinnings in statistics. Since the logistic regression model predicts probabilities, given a particular logistic model we can ask, "What is the probability that this model produced the set of observed outcomes y?" We might naturally adjust the parameters of our model until the probability of drawing our dataset from the model is as high as possible. Although we will not prove it in this section, this procedure is equivalent to minimizing the cross entropy cost—this is the *maximum likelihood* statistical justification for the cross entropy cost.

Gradient of the Cross Entropy Cost¶

In order to run gradient descent on the cross entropy cost we must calculate the gradient of the cost function. First, we compute the derivative of the sigmoid function since we'll use it in our gradient calculation.

$$ \begin{aligned} \sigma(t) &= \frac{1}{1 + e^{-t}} \\ \sigma'(t) &= \frac{e^{-t}}{(1 + e^{-t})^2} \\ \sigma'(t) &= \frac{1}{1 + e^{-t}} \cdot \left(1 - \frac{1}{1 + e^{-t}} \right) \\ \sigma'(t) &= \sigma(t) (1 - \sigma(t)) \end{aligned} $$

The derivative of the sigmoid function can be conveniently expressed in terms of the sigmoid function itself.

As a shorthand, we define $\sigma_i = f_\hat{\theta}(X_i) = \sigma(X_i \cdot \hat \theta)$. We will soon need the gradient of σ_i with respect to the vector $\hat{\theta}$ so we will derive it now using a straightforward application of the chain rule.

$$ \begin{aligned} \nabla_{\hat{\theta}} \sigma_i &= \nabla_{\hat{\theta}} \sigma(X_i \cdot \hat \theta) \\ &= \sigma(X_i \cdot \hat \theta) (1 - \sigma(X_i \cdot \hat \theta)) \nabla_{\hat{\theta}} (X_i \cdot \hat \theta) \\ &= \sigma_i (1 - \sigma_i) X_i \end{aligned} $$

Now, we derive the gradient for the cross entropy cost with respect to the model parameters $\hat{\theta}$. In the derivation below, we let $\sigma_i = f_\hat{\theta}(X_i) = \sigma(X_i \cdot \hat \theta)$.

$$ \begin{aligned} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_i \left(- yi \ln (f\hat{\theta}(X_i)) - (1 - yi) \ln (1 - f\hat{\theta}(X_i) \right) \ &= \frac{1}{n} \sum_i \left(- y_i \ln \sigma_i - (1 - y_i) \ln (1 - \sigmai) \right) \ \nabla{\hat{\theta}} L(\hat{\theta}, X, y) &= \frac{1}{n} \sum_i \left(

```
- \frac{y_i}{\sigma_i} \nabla_{\hat{\theta}} \sigma_i
+ \frac{1 - y_i}{1 - \sigma_i} \nabla_{\hat{\theta}} \sigma_i
```

\right) \ &= - \frac{1}{n} \sum_i \left(\frac{y_i}{\sigma_i} - \frac{1 - y_i}{1 - \sigmai} \right) \nabla{\hat{\theta}} \sigma_i \ &= - \frac{1}{n} \sum_i \left(\frac{y_i}{\sigma_i} - \frac{1 - y_i}{1 - \sigma_i} \right) \sigma_i (1 - \sigma_i) X_i \ &= - \frac{1}{n} \sum_i \left(y_i - \sigma_i \right) X_i \ \end{aligned} $$

The surprisingly simple gradient expression allows us to fit a logistic model to the cross entropy loss using gradient descent.

Summary¶

The cross entropy cost that we use to fit the logistic model. Since it is a convex function, we use gradient descent to fit the model to the cost. We now have the necessary components of logistic regression: the model, cost function, and minimization procedure.

Show Widgets Open on DataHub

Table of Contents

Using Logistic Regression¶

We have developed all the components of logistic regression. First, the logistic model used to predict probabilities:

$$ \begin{aligned} f_\hat{\theta} (x) = \sigma(\hat{\theta} \cdot x) \end{aligned} $$

Then, the cross entropy cost function:

$$ \begin{aligned} L(\hat{\theta}, X, y) = &= \frac{1}{n} \sum_i \left(- y_i \ln \sigma_i - (1 - y_i) \ln (1 - \sigma_i) \right) \\ \end{aligned} $$

Finally, the gradient of the cross entropy cost for gradient descent:

$$ \begin{aligned} \nabla_{\hat{\theta}} L(\hat{\theta}, X, y) &= - \frac{1}{n} \sum_i \left(y_i - \sigma_i \right) X_i \\ \end{aligned} $$

In the expressions above, we let $ X $ represent the $ n \times p $ input data matrix, $ y $ the vector of observed data values, and $ f_\hat{\theta}(x) $ the logistic model. As a shorthand, we define $ \sigma_i = f_\hat{\theta}(X_i) = \sigma(X_i \cdot \hat \theta) $.

Logistic Regression on LeBron Shots¶

Let us now return to the problem we faced at the start of this chapter: predicting which shots LeBron James will make. We start by loading the dataset of shots taken by LeBron in the 2017 NBA Playoffs.

```
lebron = pd.read_csv('lebron.csv')
lebron
```

	game_date	minute	opponent	action_type	shot_type	shot_dista
0	20170415	10	IND	Driving Layup Shot	2PT Field Goal	0
1	20170415	11	IND	Driving Layup Shot	2PT Field Goal	0
2	20170415	14	IND	Layup Shot	2PT Field Goal	0
...	...	...	...	...	...	...
381	20170612	46	GSW	Driving Layup Shot	2PT Field Goal	1
382	20170612	47	GSW	Turnaround Fadeaway shot	2PT Field Goal	14
383	20170612	48	GSW	Driving Layup Shot	2PT Field Goal	2

384 rows × 7 columns

We've included a widget below to allow you to pan through the entire DataFrame.

```
df_interact(lebron)
```

Show Widget

```
(384 rows, 7 columns) total
```

We start by using using only the shot distance to predict the shot make or miss. `scikit-learn` conveniently provides a logistic regression classifier as the `sklearn.linear_model.LogisticRegression` class. To use the class, we first create our data matrix `X` and vector of observed outcomes `y`.

```
X = lebron[['shot_distance']].as_matrix()
y = lebron['shot_made'].as_matrix()
print('X:')
print(X)
print()
print('y:')
print(y)
```

```
X:
[[ 0]
 [ 0]
 [ 0]
 ...
 [ 1]
 [14]
 [ 2]]

y:
[0 1 1 ... 1 0 1]
```

As is customary, we split our data into a training set and a test set.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=40, random_state=42
)
print(f'Training set size: {len(y_train)}')
print(f'Test set size: {len(y_test)}')
```

```
Training set size: 344
Test set size: 40
```

`scikit-learn` makes it simple to initialize the classifier and fit it on `X_train` and `y_train`:

```
from sklearn.linear_model import LogisticRegression
simple_clf = LogisticRegression()
simple_clf.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr',
n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear',
tol=0.0001,
          verbose=0, warm_start=False)
```

To visualize the classifier's performance, we plot the original points and the classifier's predicted probabilities.



Evaluating the Classifier¶

One method to evaluate the effectiveness of our classifier is to check its prediction accuracy: what proportion of points does it predict correctly?

```
simple_clf.score(X_test, y_test)
```

```
0.6
```

Our classifier achieves a rather low accuracy of 0.60 on the test set. If our classifier simply guessed each point at random, we would expect an accuracy of 0.50. In fact, if our classifier simply predicted that every shot LeBron takes will go in, we would also get an accuracy of

0.60:

```
# Calculates the accuracy if we always predict 1
np.count_nonzero(y_test == 1) / len(y_test)
```

```
0.6
```

For this classifier we only used one out of several possible features. As in multivariable linear regression, we will likely achieve a more accurate classifier by incorporating more features.

Multivariable Logistic Regression¶

Incorporating more numerical features in our classifier is as simple as extracting additional columns from the `lebron` DataFrame into the `X` matrix. Incorporating categorical features, on the other hand, requires us to apply a one-hot encoding. In the code below, we augment our classifier with the `minute`, `opponent`, `action_type`, and `shot_type` features, using the `DictVectorizer` class from `scikit-learn` to apply a one-hot encoding to the categorical variables.

```
from sklearn.feature_extraction import DictVectorizer

columns = ['shot_distance', 'minute', 'action_type',
'shot_type', 'opponent']
rows = lebron[columns].to_dict(orient='row')

onehot = DictVectorizer(sparse=False).fit(rows)
X = onehot.transform(rows)
y = lebron['shot_made'].as_matrix()

X.shape
```

```
(384, 42)
```

We will again split the data into a training set and test set:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=40, random_state=42
)
print(f'Training set size: {len(y_train)}')
print(f'Test set size: {len(y_test)}')
```

```
Training set size: 344
Test set size: 40
```

Finally, we fit our model once more and check its accuracy:

```
clf = LogisticRegression()
clf.fit(X_train, y_train)
print(f'Test set accuracy: {clf.score(X_test, y_test)}')
```

```
Test set accuracy: 0.725
```

This classifier is around 12% more accurate than the classifier that only took the shot distance into account.

Summary¶

We have developed the mathematical and computational machinery needed to use logistic regression for classification. Logistic regression is widely used for its simplicity and effectiveness in prediction.

Reference Tables

This appendix contains reference tables for the `pandas`, `seaborn`, `matplotlib`, and `scikit-learn` methods used in the book. It is meant to provide a helpful overview of the small subset of methods that we use most often in this book.

For each library, we list the methods used, the chapter where each method is first mentioned, and a brief description of the method's functionality.

pandas

Function	Chapter	Description
`pd.DataFrame(data)`	Tabular Data and pandas	Create a DataFrame from a two-dimensional array or dictionary `data`
`pd.read_csv(filepath)`	Tabular Data and pandas	Import a CSV file from `filepath` as a pandas DataFrame
`pd.DataFrame.head(n=5)` `pd.Series.head(n=5)`	Tabular Data and pandas	View the first `n` rows of a DataFrame or Series
`pd.DataFrame.index` `pd.DataFrame.columns`	Tabular Data and pandas	View a DataFrame's index and column values
`pd.DataFrame.describe()` `pd.Series.describe()`	Exploratory Data Analysis	View descriptive statistics about a DataFrame or Series
`pd.Series.unique()`	Exploratory Data Analysis	View unique values in a Series
`pd.Series.value_counts()`	Exploratory Data Analysis	View the number of times each unique value appears in a Series
`df[col]`	Tabular Data and pandas	From DataFrame `df`, return column `col` as a Series
`df[[col]]`	Tabular Data and pandas	From DataFrame `df`, return column `col` as a DataFrame
`df.loc[row, col]`	Tabular Data and pandas	From DataFrame `df`, return rows with index name `row` and column name `col`; `row` can alternatively be a boolean Series
`df.iloc[row, col]`	Tabular Data and pandas	From DataFrame `df`, return rows with index number `row` and column number `col`; `row` can alternatively be a boolean Series
`pd.DataFrame.isnull()`	Data	View missing values in a

	Cleaning	DataFrame or Series
`pd.DataFrame.fillna(value)` `pd.Series.fillna(value)`	Data Cleaning	Fill in missing values in a DataFrame or Series with `value`
`pd.DataFrame.dropna(axis)` `pd.Series.dropna()`	Data Cleaning	Drop rows or columns with missing values from a DataFrame or Series
`pd.DataFrame.drop(labels, axis)`	Data Cleaning	Drop rows or columns named `labels` from DataFrame along `axis`
`pd.DataFrame.rename()`	Data Cleaning	Rename specified rows or column in DataFrame
`pd.DataFrame.replace(to_replace, value)`	Data Cleaning	Replace `to_replace` values with `value` in DataFrame
`pd.DataFrame.reset_index(drop=False)`	Data Cleaning	Reset a DataFrame's indices; by default, retains old indices as a new column unless `drop=True` specified
`pd.DataFrame.sort_values(by, ascending=True)`	Tabular Data and pandas	Sort a DataFrame by specified columns `by`, in ascending order by default
`pd.DataFrame.groupby(by)`	Tabular Data and pandas	Return a GroupBy object that contains a DataFrame grouped by the values in the specified columns `by`
`GroupBy.<function>`	Tabular Data and pandas	Apply a function `<function>` to each group in a GroupBy object `GroupBy`; e.g. `mean()`, `count()`
`pd.Series.<function>`	Tabular Data and pandas	Apply a function `<function>` to a Series with numerical values; e.g. `mean()`, `max()`, `median()`
`pd.Series.str.<function>`	Tabular Data and pandas	Apply a function `<function>` to a Series with string values; e.g. `len()`, `lower()`, `split()`
`pd.Series.dt.<property>`	Tabular Data and pandas	Extract a property `<property>` from a Series with Datetime values; e.g. `year`, `month`, `date`
`pd.get_dummies(columns, drop_first=False)`	---	Convert categorical variables `columns` to dummy variables; default retains all variables unless `drop_first=True`

		specified
`pd.merge(left, right, how, on)`	Exploratory Data Analysis; Databases and SQL	Merge two DataFrames `left` and `right` together on specified columns `on` ; type of join depends on `how`
`pd.read_sql(sql, con)`	Databases and SQL	Read a SQL query `sql` on a database connection `con` , and return result as a pandas DataFrame

Seaborn

Function	Chapter	Description
`sns.lmplot(x, y, data, fit_reg=True)`	Data Visualization	Create a scatterplot of `x` versus `y` from DataFrame `data`, and by default overlay a least-squares regression line
`sns.distplot(a, kde=True)`	Data Visualization	Create a histogram of `a`, and by default overlay a kernel density estimator
`sns.barplot(x, y, hue=None, data, ci=95)`	Data Visualization	Create a barplot of `x` versus `y` from DataFrame `data`, optionally factoring data based on `hue`, and by default drawing a 95% confidence interval (which can be turned off with `ci=None`)
`sns.countplot(x, hue=None, data)`	Data Visualization	Create a barplot of value counts of variable `x` chosen from DataFrame `data`, optionally factored by categorical variable `hue`
`sns.boxplot(x=None, y, data)`	Data Visualization	Create a boxplot of `y`, optionally factoring by categorical variables `x`, from the DataFrame `data`
`sns.kdeplot(x, y=None)`	Data Visualization	If `y=None`, create a univariate density plot of `x`; if `y` is specified, create a bivariate density plot
`sns.jointplot(x, y, data)`	Data Visualization	Combine a bivariate scatterplot of `x` versus `y` from DataFrame `data`, with univariate density plots of each variable overlaid on the axes
`sns.violinplot(x=None, y, data)`	Data Visualization	Draws a combined boxplot and kernel density estimator of variable `y`, optionally factored by categorical variable `x`, chosen from DataFrame `data`

matplotlib

Types of Plots

Function	Chapter	Description
`plt.scatter(x, y)`	Data Visualization	Creates a scatter plot of the variable x against the variable y
`plt.plot(x, y)`	Data Visualization	Creates a line plot of the variable x against the variable y
`plt.hist(x, bins=None)`	Data Visualization	Creates a histogram of x. Bins argument can be an integer or sequence
`plt.bar(x, height)`	Data Visualization	Creates a bar plot. `x` specifies x-coordinates of bars, `height` specifies heights of the bars
`plt.axvline(x=0)`	Data Visualization	Creates a vertical line at the x value specified
`plt.axhline(y=0)`	Data Visualization	Creates a horizontal line at the y value specified

Plot additions

Function	Chapter	Description
`%matplotlib inline`	Data Visualization	Causes output of plotting commands to be displayed inline
`plt.figure(figsize=(3, 5))`	Data Visualization	Creates a figure with a width of 3 inches and a height of 5 inches
`plt.xlim(xmin, xmax)`	Data Visualization	Sets the x-limits of the current axes
`plt.xlabel(label)`	Data Visualization	Sets an x-axis label of the current axes
`plt.title(label)`	Data Visualization	Sets a title of the current axes
`plt.legend(x, height)`	Data Visualization	Places a legend on the axes
`fig, ax = plt.subplots()`	Data Visualization	Creates a figure and set of subplots
`plt.show()`	Data Visualization	Displays a figure

scikit-learn

Models and Model Selection

Import	Function	Section	Description
`sklearn.model_selection`	`train_test_split(*arrays, test_size=0.2)`	Modeling and Estimation	Returns two random subsets of each array passed in, with 0.8 of the array in the first subset and 0.2 in the second subset
`sklearn.linear_model`	`LinearRegression()`	Modeling and Estimation	Returns an ordinary least squares Linear Regression model
`sklearn.linear_model`	`LassoCV()`	Modeling and Estimation	Returns a Lasso (L1 Regularization) linear model with picking the best model by cross validation
`sklearn.linear_model`	`RidgeCV()`	Modeling and Estimation	Returns a Ridge (L2 Regularization) linear model with picking the best model by cross validation
`sklearn.linear_model`	`ElasticNetCV()`	Modeling and Estimation	Returns a ElasticNet (L1 and L2 Regularization) linear model with picking the best model by cross

			validation
`sklearn.linear_model`	`LogisticRegression()`	Modeling and Estimation	Returns a Logistic Regression classifier
`sklearn.linear_model`	`LogisticRegressionCV()`	Modeling and Estimation	Returns a Logistic Regression classifier with picking the best model by cross validation

Working with a Model

Assuming you have a `model` variable that is a `scikit-learn` object:

Function	Section	Description
`model.fit(X, y)`	Modeling and Estimation	Fits the model with the X and y passed in
`model.predict(X)`	Modeling and Estimation	Returns predictions on the X passed in according to the model
`model.score(X, y)`	Modeling and Estimation	Returns the accuracy of X predictions based on the corect values (y)

Contributors

This textbook contains substantial contributions from Data 100 students. We list the contributors below and thank them for their effort in creating content for the textbook.

Name	Contributions
Ananth Agarwal	9.1 (Relational Databases), 9.2 (SQL Queries), 13.3 (Cross-Validation)
Ashley Chien	9.3 (SQL Joins), Reference Table Appendix
Andrew Do	8.1 (Python String Methods), 8.2 (Regular Expressions), 8.3 (Regex in Python and pandas)
Tiffany Jann	7.1 (HTTP), 11.3 (Convexity)
Andrew Kim	7.1 (HTTP), 8.1 (Python String Methods), 8.2 (Regular Expressions), 8.3 (Regex in Python and pandas)
Jun Seo Park	9.1 (Relational Databases), 9.2 (SQL Queries), Reference Table Appendix
Allen Shen	11.3 (Convexity), 14.1 (Expectation and Variance)
Katherine Yen	9.3 (SQL Joins), 13.3 (Cross-Validation)
Daniel Zhu	8.1 (Python String Methods), 8.2 (Regular Expressions), 8.3 (Regex in Python and pandas), 14.1 (Expectation and Variance)